Design Time and Run Time Formal Safety Analysis using Executable Models

Axel Habermaier

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Fakultät für Angewandte Informatik

5. September 2016

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Alexander Knapp
Tag der mündlichen Prüfung:	21. November 2016

Für Karin.

Danksagung

Diese Dissertation wäre ohne die vielseitige Unterstützung aus meinem privaten und beruflichen Umfeld wohl kaum möglich gewesen. Ich möchte die Gelegenheit daher nutzen, um mich bei den folgenden Leuten herzlich zu bedanken.

Für die Betreuung dieser Arbeit danke ich Herrn *Prof. Dr. Wolfgang Reif*, der mir einerseits das Vertrauen geschenkt und die Möglichkeiten gegeben hat, etwas abseits der etablierten Wege zu forschen, andererseits aber das Abdriften in falsche Richtungen immer rechtzeitig verhindern konnte. Daher bedanke ich mich für die vielen guten Ratschläge in zahlreichen kreativen Diskussionen, die maßgeblich zur Entstehung dieser Arbeit beigetragen haben.

Auch Herrn *Prof. Dr. Alexander Knapp* möchte ich ein herzliches Dankeschön aussprechen, hat er doch mit seinen tiefen Einblicken in die Welt der formalen Methoden entscheidende Hinweise, Anmerkungen und Anregungen zu den formalen Grundlagen sowie des gesamten Ansatzes beigetragen. Aber auch bei allen Belangen zum Thema LaTeX war sein Rat stets von unschätzbarem Wert.

Viele Ergebnisse dieser Arbeit sind in Zusammenarbeit mit den Kollegen am Institut für Software & Systems Engineering entstanden; viele dieser Kollegen sind inzwischen Freunde geworden. Ich bedanke mich bei allen für die vielen Anregungen auf unseren zahlreichen Oberseminaren und Lehrstuhlklausuren sowie für die produktive und immer freundschaftliche Zusammenarbeit in entspannter Atmosphäre. Die freigesetzte kreative Energie unter anderem beim Mittagessen war ein wohltuendes Kontrastprogramm zu den üblichen Tätigkeiten in Forschung und Lehre. Wertschätzung!

Besonderer Dank gilt den Kollegen *Benedikt Eberhardinger*, *Gidon Ernst*, *Hella Seebach*, *Johannes Leupolz*, *Alwin Hoffmann* und *Gerhard Schellhorn* für die vielen kreativen Ansatzpunkte, Ideen und hilfreichen Diskussionen, die großen Einfluss auf die Inhalte der folgenden Seiten hatten. Auch *Dominik Klumpp* und *Birgit Kühbacher* gebührt Dank für ihr großes Engagement im Rahmen ihrer Bachelorarbeiten, mit denen sie meine Forschung in vielerlei Hinsicht unterstützen konnten.

Für das intensive und aufmerksame Korrekturlesen dieser umfangreichen Arbeit bedanke ich mich bei *Andreas Angerer*, *Benedikt Eberhardinger*, *Johannes Leupolz*, *Hella Seebach* und *Alexander Schiendorfer*.

Zu guter Letzt möchte ich noch meiner gesamten Familie danken, die während des langen Weges hin zur vorliegenden Dissertation nicht nur immerzu unerschütterlich an mich geglaubt hat, sondern mir stets auch den Rücken freihielt und die dazu nötige Geduld aufbringen konnte. Vielen Dank dafür! Insbesondere gilt dies natürlich auch für meine Frau *Karin Habermaier*, die mir stets zur Seite stand, mich motivierte und entlastete wo immer nötig und möglich. Ohne sie hätte ich wohl kaum die Kraft gehabt, diese Arbeit zu Ende zu führen. Vielen Dank für das entgegengebrachte Verständnis, die unendliche Geduld und das unermüdliche Vertrauen!

Axel Habermaier

Abstract

Safety-critical systems can be negatively affected by faults such as component failures or environmental disturbances. One or more occurrences of such faults might lead to hazards, that is, potentially disastrous situations or conditions that may cause accidents with economical losses, environmental damage, injuries, or loss of lives. Safety analysis is the process of identifying such hazards as well as their root causes in order to assess a system's overall safety. Traditional safety analysis techniques such as Fault Tree Analysis or Failure Modes and Effects Analysis help to systematically assess system safety by informally dissecting the system's behavior and component interdependencies. However, the increasing complexity of safety-critical systems, in part due to the introduction of more and more software-based components, requires more sophisticated safety analysis techniques to thoroughly analyze system behavior with regard to faults. Additionally, traditional safety analysis techniques are not suited for the analysis of self-organizing systems, an emerging class of often safety-critical systems that change their behavior and structure during operation in a way that cannot be predicted during system development. Self-organization thus necessitates new safety analysis approaches that are at least partially conducted at run time while the system is already in operation.

The contribution of this thesis is a systematic design time and run time modeling and analysis approach for safety-critical systems based on formal methods and executable models. Its main achievements are threefold: Firstly, a systematic modeling approach and the executable modeling language S# are introduced. Secondly, a safety analysis technique based on explicit-state model checking and its formal foundations are defined, increasing analysis efficiency and practical usability compared to alternative techniques. Thirdly, a unified model execution and analysis approach is used to simulate and visualize S# models and to systematically conduct design time and run time safety analyses for them with consistent semantics. The contributions are evaluated using different case studies from multiple application domains.

Systematic Modeling with the Executable Modeling Language S#. The systematic modeling approach for safety-critical systems brings forward well-established software engineering principles and best practices to the modeling and analysis of safety-critical systems during all phases of development. It fosters the systematic development of high-quality, comprehensible, adequate, and modular models of such systems, while also allowing the use of formal safety analysis techniques for rigorous safety assessment. In doing so, the approach and the executable modeling language S# facilitate safety analyses throughout the entire system life-cycle, modeling and safety assessment of design alternatives, as well as model simulation, model visualization, model debugging, and model-based testing. The S# modeling language and its analysis framework hide the details of the underlying formal analysis techniques while providing flexible modeling composition and instantiation capabilities to conveniently express and configure different system configurations for formal safety analyses. This flexibility helps with exploring the design space of a safety-critical system early in its development and also forms the basis for the run time analysis approach of self-organizing systems.

Formalization of an Efficient Formal Safety Analysis Technique. The S# framework uses Deductive Cause Consequence Analysis (DCCA) as its formal, model-based safety analysis technique for sound reasoning about the safety-critical aspects of a system with mathematical rigor. DCCA employs exhaustive model checking to compute how faults (the causes) such as component failures or environmental disturbances can cause hazards (the consequences): From a model of a safety-critical system that describes the system's nominal behavior as well as the relevant faults, DCCA determines all minimal critical fault sets, i.e., combinations of faults that can cause hazards, allowing the evaluation of the system's overall safety. In order to increase the efficiency of DCCA and the S# framework's explicit-state model checking approach, fault-aware modeling and specification formalisms are introduced that decrease explicit-state analysis times by up to three orders of magnitude. Additionally, conceptual improvements of DCCA enhance its model checking workflow as well as its applicability, allowing it to be used with a broader variety of analysis tools besides S#. There also is a new execution variant for DCCA that reduces analysis times in many cases by taking advantage of explicit-state model checking: Irrelevant faults can be removed when checking a specific fault set for criticality, allowing multiple significantly smaller models to be checked instead of a single large one. Furthermore, heuristics are developed that help to reduce the number of criticality checks that have to be carried out by a complete DCCA, as DCCA's worst case execution time is exponential in the number of analyzed faults.

Unified Model Execution for Design Time & Run Time Analysis. The S# modeling and analysis framework for safety-critical systems conducts DCCAs fully automatically using a unified model execution approach based on explicit-state model checking. The executability of S# models results from S#'s foundation in the ISO-standardized C# programming language and the .NET run time environment. Instead of relying on model transformations like most other safety analysis tools, S# unifies simulations, model-based tests, visualizations, fully exhaustive model checking, and formal safety analyses by executing the models with consistent semantics regardless of whether a simulation is run or some formula is model checked. Consequently, S# models are allowed to use most advanced modeling language features inherited from C# such as inheritance, generics, virtual dispatch, lambda functions, and so on. Standard C# tools like Visual Studio can be used for model development and analysis; in particular, models can be debugged even while model checking. S#'s flexible model composition and instantiation capabilities also allow for run time safety analyses of self-organizing systems whose safety cannot be exhaustively assessed during system development. Their typically infinite amount of possible system configurations requires safety analyses to be conducted at run time using a virtual commissioning process: Whenever a self-organizing system changes its structure or behavior, safety analyses are conducted for the system's actual configuration either before or while the system continues its operation using the newly computed system configuration.

Contents

Summary. This overview of safety-critical systems and model-based safety analysis motivates the goals of this thesis and its main contributions. Systems are safety-critical when they have the potential to cause situations resulting in environmental damage, economic losses, injuries, or loss of lives. Model-based safety analysis techniques use formal methods to determine the combinations of faults that lead to such situations with mathematical rigor. The adequacy of the results obtained with formal safety analyses depends on the adequacy of the analyzed models, which can be improved by following a systematic modeling approach. Additionally, the class of self-organizing systems poses additional challenges for formal safety analyses due to their high level of redundancy and run time reconfiguration capabilities.

1 Overview and Motivation

Failures of safety-critical systems result in hazards, i.e., potentially disastrous situations or conditions that may lead to accidents causing economical losses, environmental damage, injuries, or deaths [106]. Safety-critical systems have to anticipate various faults [9] that inevitably occur because of unexpected environmental conditions, deliberate or non-deliberate human-made incorrect operational decisions, mistakes made during system development, or failing hardware and software components that do not behave as specified, etc. While safety-critical systems are designed to prevent hazards for as long as possible, various accidents throughout recent history such as the Therac 25 accidents or the Fukushima Daiichi disaster [100, 132] clearly show that hazards eventually do in fact occur. Consequently, there are always risks associated with building, operating, and maintaining safety-critical systems. These risks must be justified, however, in the sense that the money and effort required to further reduce the risks are greatly disproportionate with the potential risk reductions [25]. The concepts of acceptable and tolerable risks are based on social consent and are often regulated by international norms and standards for the development of safety-critical systems such as IEC 61508 or IEC 61511 [103, 104], among many others [57].

Complex system designs and complicated operational procedures have adverse effects on system safety as they increase the likelihood of development faults, analysis oversights, and operational faults [131, 186, 200]. Still, safety-critical systems continue to become more complex, partly due to the increasing use of software [25]. Software-based implementations of system functionality are often more flexible and more cost effective than hardware implementations, broadening the scope and the field of operations of safety-critical systems. However, software development and the microprocessor infras-

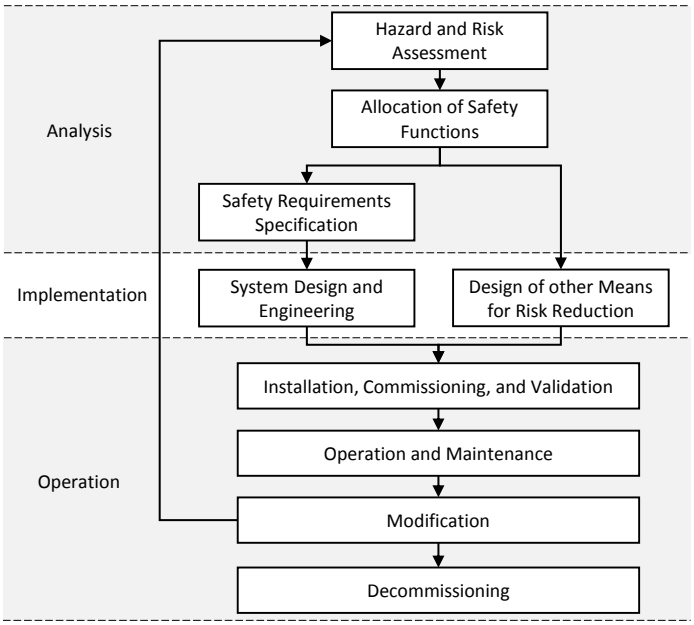


Figure 1.1. Simplified overview of the three main phases of the IEC 61511 safety life-cycle [103, 182]: During the analysis phase, hazards are identified and their causes and risks are determined. Additionally, safety functions are allocated to individual parts of the system under development and safety requirements are specified and documented. In the implementation phase, the system is designed, software is implemented, and hardware is fabricated; potentially, additional risk-reducing safety measures are designed to complement the system. The final phase is operation, where the system is installed, commissioned, and validated. Maintenance is typically required during operation until the system is eventually decommissioned. Modifications of the system necessitate additional safety analyses and implementation work.

tructure required to run software are complex and error-prone: The size of NASA's flight control software, for instance, has been shown to grow roughly exponentially over the years, similar to the software of military aircraft and automobiles [200].

Figure 1.1 gives an overview of the life-cycle of safety-critical systems from conceptual inception, safety analysis, design and implementation, to operation, maintenance, and eventual decommissioning. This thesis focuses on the task of safety assessment; the terms safety assessment and safety analysis are generally used interchangeably. It is assumed that the hazards of the safety-critical system to be analyzed are already identified and that all potentially relevant faults are known. Traditional safety analysis techniques such as Failure Modes and Effects Analysis (FMEA), for instance, can be used to identify the hazards of a safety-critical system [25, 55, 166], while failure modes of off-the-shelf components are usually provided by the component vendors. In addition to FMEA, Fault Tree Analysis (FTA), Hazard and Operability Study, Event Tree Analysis, and many others allow for systematic safety analyses [25, 55, 88, 193] by identifying possible combinations of faults that might result in hazards. Therefore, the techniques help to design and integrate safety measures that reduce risks by making hazards less likely or by reducing their severity.

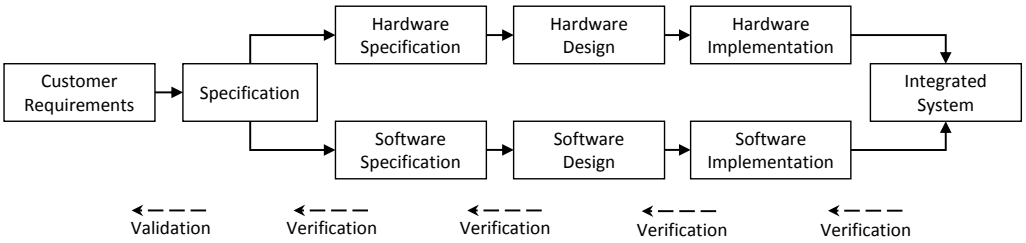


Figure 1.2. Validation and verification during the development process [186]: While the software and hardware integration is typically only tested and not formally verified, implementations of safety-critical components can be shown to adhere to their designs and specifications using formal verification techniques. A validation against the actual customer requirements shows the adequacy of the specification and the developed system. No formal validation techniques exist, although formal specification and modeling often help to improve adequacy indirectly [10].

Many commonly used traditional safety analysis techniques such as FMEA and FTA were developed in the middle of the last century, in a time when software did not yet play an important role or was not used at all [166, 193]. Additionally, FMEAs and FTAs are typically performed manually, relying on human experience and the ability of a human to understand and predict system behavior. Consequently, there is a high risk for incomplete, inconsistent, or erroneous analyses. Recent improvements such as Component Fault Trees [74] help to tackle the increasing complexity; for software-intensive safety-critical systems, in particular, new model-based safety analysis techniques rooted in formal methods have been developed [82, 113, 135, 151]. This thesis enhances these formal techniques, formalizing core concepts in the area of safety analysis such as fault injection and fault removal and improving the applicability and efficiency of the underlying modeling formalisms and analysis mechanisms. Moreover, a modeling and analysis framework for safety-critical systems using these formal techniques is introduced that is based on executable models, thereby also allowing flexible model composition for run time safety analyses of self-organizing systems that are not possible with current analysis approaches.

1.1 Formal Safety Analysis

In order to deal with the increasing complexity of systems and software, formal methods allow for logical reasoning about software and system designs as well as implementations with mathematical rigor [13]. Mathematical specifications of system requirements, architectures, and designs improve the likeliness of finding and identifying flaws, misunderstandings, and inconsistencies before the software is actually written and the system is assembled, thus reducing the cost to fix these issues [10]. However, formal methods guarantee neither adequacy nor validity of the analyses, as it is impossible to decide whether the analyzed system is adequate, that is, whether it fulfills the needs of the people affected by its operation and use. Consequently, the advantage of using formal methods is not the guarantee of correctness, but the increased confidence in the correctness and adequacy of the final product as illustrated by Figure 1.2. Moreover, development costs are potentially reduced as more flaws are found in earlier stages of

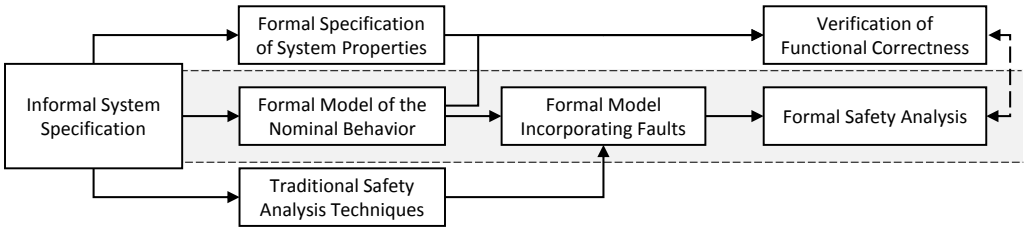


Figure 1.3. Relationship between traditional and formal safety analysis as well as verification of functional correctness [80]: From an informal specification of the safety-critical system that is to be developed, a formal model of the system's nominal behavior is created; together with formal specifications of the properties the system should comply with, functional correctness can be verified. For the hazards identified with traditional safety analysis techniques such as FMEA, formal safety analyses are conducted based on the formal model of the nominal behavior that is extended to incorporate the presumably relevant faults. Formal safety analysis subsumes functional correctness in the sense that safety analyses label functionally incorrect systems as inherently unsafe, i.e., hazards can occur even without any faults.

the development process where countermeasures are less costly. The downside is the need for engineers experienced with formal methods and the time it takes to formally specify and analyze complex systems. Thus, a trade-off must be found to balance these advantages and disadvantages.

Formal, model-based safety analysis techniques apply formal methods to the field of safety assessment, automatically computing all minimal critical fault sets of safety-critical systems [80, 113]. These fault sets represent combinations of faults that can cause hazards. Deductive Cause Consequence Analysis (DCCA) is one such technique [159], based on model checking, that rigorously determines the relationships between faults (the causes) and hazards (the consequences). DCCAs are conducted automatically by safety analysis tools such as the S# framework or VECS [82, 135], for instance; the FSAP/Compass toolsets [26, 27] and AltaRica [15] perform similar safety analyses. Formal safety analysis is based on formal models of the systems to be analyzed, increasing consistency and improving adequacy over traditional safety analysis techniques. Additionally, completeness can be guaranteed, hence formal safety analysis tools find all minimal critical fault sets for all analyzed hazards. Model checking-based safety analysis techniques also generate witnesses that explain how critical fault sets can cause a hazard, that is, under which circumstances the faults in the critical set can result in potentially dangerous and harmful situations. In other words, a counterexample is generated disproving a fault set as safe, providing hints for possible system design improvements that potentially reduce safety risks. Figure 1.3 illustrates the integration of formal safety analysis with functional correctness verification and traditional safety analysis techniques.

Many successful real-world applications of formal methods [25, 90, 134, 206] show the potential benefits of using formal modeling, specification, and verification techniques for the development of safety-critical systems. Before the launch of NASA's Deep Space 1 spacecraft [89], for instance, the SPIN model checker [98] was able to identify five concurrency errors in a model of the deep space autonomy flight software that had

remained undetected despite extensive testing. After launch, an isomorphic bug caused a deadlock in another part of the software that had not been model checked and that also was not found by tests; a retrospective analysis with SPIN was again able to identify and diagnose the problem.

Consequently, many international standards like IEC 61508 [104] for safety-critical software development, EN 50128 [59] in the railway sector, ISO 26262 [107] in the automotive sector, or DO-178C [174] in the avionics domain highly recommend the use of formal methods for system design and safety analysis in order to reach the highest safety levels [11, 71]. Additionally, formal techniques can be used to document the safety assessment steps performed during the entire life-cycle of a safety-critical system as required by IEC 61508 or EN 50128. As compliance with these standards is often required either by law or contractually as well as due to recent usability improvements in both formal techniques and tools, industry projects increasingly rely on formal methods. A survey conducted by Woodcock et al. [206], for instance, shows that formal specification and formal modeling as well as model checking are among the techniques most often employed in the industry. In particular, the use of model checking has increased from 13% to 51% within the last decade across all examined industry projects. Formal methods are reported to have an overall positive effect on development time, cost, and quality, with no participants of the survey reporting quality reductions.

1.2 Executable Models of Safety-Critical Systems

All model-based analysis techniques have one common weakness: Analysis results are only adequate if the underlying system models are adequate. If the models do not appropriately represent the real systems, the results obtained from formal analyses do not necessarily reflect the systems' actual properties. Adequate models for formal safety analyses are even harder to create than models used for formal correctness verification only: The systems' behavior in the case of faults must be considered and modeled in addition to the desired functional behavior. Consequently, specifically tailored modeling language constructs for fault modeling as well as a systematic modeling methodology are required in order to support the development of adequate models.

Various modeling languages and formalisms with different levels of expressiveness exist, ranging from low-level Kripke structures to high-level SysML models [13, 153]. The language that fits best usually depends on the system that is to be modeled and analyzed. The Compass and VECS safety analysis tools [135, 151], for instance, use an extended subset of AADL or the custom SAML language, respectively. Alternatively, there are models authored in an executable modeling language that, by contrast, can immediately be executed just like regular programs without requiring any intermediate model transformations. SystemC or S# [73, 82] are examples of such executable formalisms; the latter is used throughout this thesis. As illustrated by Figure 1.4, S# models readily support model simulations, model visualizations, as well as explicit-state model checking due to their executability.

Executable models can often be integrated with explicit-state model checkers such as LTSmin [116], provided that the models have a finite number of reachable states only.

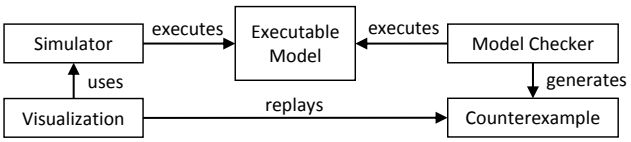


Figure 1.4. Executable models such as the ones authored in S# do not require any complex model transformations for simulations or model checking; instead, the models can be directly executed like regular programs. Visualizations make use of simulations to allow for graphical inspections of system states and behavior; they additionally support replays of counterexamples or witnesses generated by the model checker.

Symbolic model checking of executable models, on the other hand, can be problematic due to complex model transformations that are typically necessary [64]. Model simulations and visualizations are useful to validate the adequacy of the analyzed models or to view, debug, and understand witnesses for minimal critical fault sets found by a model checker [61], for example. Non-executable modeling languages like the Compass toolset's AADL dialect [151] make visualizations more technically challenging as additional model transformations or run time interpretations are necessary. Thus, the visualized model is only loosely linked to the model checked one, potentially introducing discrepancies. Executable models, by contrast, do not suffer this problem, increasing the confidence in the simulations, visualizations, and formal analyses.

Executable modeling languages weaken the distinction between programs and models to the point where both terms can often be used interchangeably [198]. In the context of safety analysis, however, the distinction is clear due to the kinds of components that the models can consist of: Some model parts represent the physical environment, hardware components, hydraulic or electrical subsystems, etc., none of which are software-based in the real system. Even those parts of the models that do in fact represent software components are generally not intended to be used as the actual implementations of the real software. The models are usually an abstraction of the real software's behavior in order to reduce the state space for model checking-based analysis techniques. Moreover, actual software implementations for safety-critical systems are typically written in the C or C++ programming languages [108, 111] for reasons of compatibility and performance. Of course, C++ programs are also models in the sense that they abstract from the bits and bytes shifted around by the microcontroller that executes the program, showing that the level of abstraction is not the distinguishing factor between models and programs.

1.3 Run Time Safety Analysis of Self-Organizing Systems

The techniques and approaches used to model, design, and evaluate safety-critical systems during development typically depend on the development phase: In the beginning during early design space exploration, for instance, highly abstract models of the desired system behavior might first be created disregarding any potential faults, using model checking to analyze the functional correctness of the system. In a second step, faults could be added to the model, using model checking to check for the minimal critical fault sets of the hazards identified for the system. Hazard mitigation strategies could

subsequently be devised from the witnesses generated by the model checker that show how minimal critical fault sets actually cause the hazards, making it necessary, for instance, to increase redundancy or to design additional error detection and recovery mechanisms. Even in this early stage, model-based safety analysis techniques facilitate the evaluation of different design alternatives of a system, making it possible to find acceptable compromises between development effort, costs, and risk reductions early in the development process [80, 82]. Once a design is chosen, the models could be gradually refined to be more concrete, potentially making the use of model checking infeasible at some point. However, models used solely for simulations and visualizations still help to develop and evaluate the systems [61]. Once actual software is written or hardware is designed, previously developed simulations and visualizations for the models could even be extended to hardware- or software-in-the-loop tests [12].

There is an important prerequisite that must be met in order to conduct safety analyses during system development: The analyzed system's structure and potential behavior must be fully known and specified at design time in order to allow model checkers to deduce the minimal critical fault sets. For safety-critical systems that are self-organizing, however, this is usually not the case: Self-organizing systems dynamically adapt their behavior and structure to changing environmental circumstances and to fault occurrences in particular, resulting in system configurations and behaviors that emerge unpredictably in a way that cannot be foreseen during development [81, 184, 202]. The primary intent of self-organizing systems is to increase system robustness with the help of a high level of redundancy and the systems' reconfiguration mechanisms. Due to the redundancy and the often infinite number of potential system configurations, it becomes impossible to conduct complete safety analyses at design time. For safety-critical self-organizing systems, it is therefore necessary to postpone model-based safety analyses to run time when more information about the actual system configurations and the emerging behavior is known. That is, safety analyses can no longer be fully conducted during system design and development, but at least partially require the system to be in operation already. Run time adaptation of the system models in accordance with the systems' self-organization mechanisms [7, 19, 48] enables model-based safety analyses for self-organizing systems, partially shifting safety analyses to run time [48, 192, 201] using efficient model checking-based safety analysis techniques.

1.4 Thesis Outline and Main Contributions

This thesis presents an approach for formal safety analyses based on executable models that can be carried out both at design time as well as at run time. The following gives a brief outline of the contents as well as the main contributions of each of the subsequent chapters; an overview of the thesis' general structure is given by Figure 1.5. Discussions of related work and evaluations of the techniques introduced throughout this thesis are generally included in those chapters they conceptually belong to. The results of this thesis are implemented in the S# formal modeling and safety analysis framework for safety-critical systems, available online in the S# repository [101]. While some of the contributions are closely interwoven with the S# framework, the formal foundations, DCCA, and the run time analysis approach for self-organizing systems are not S#-

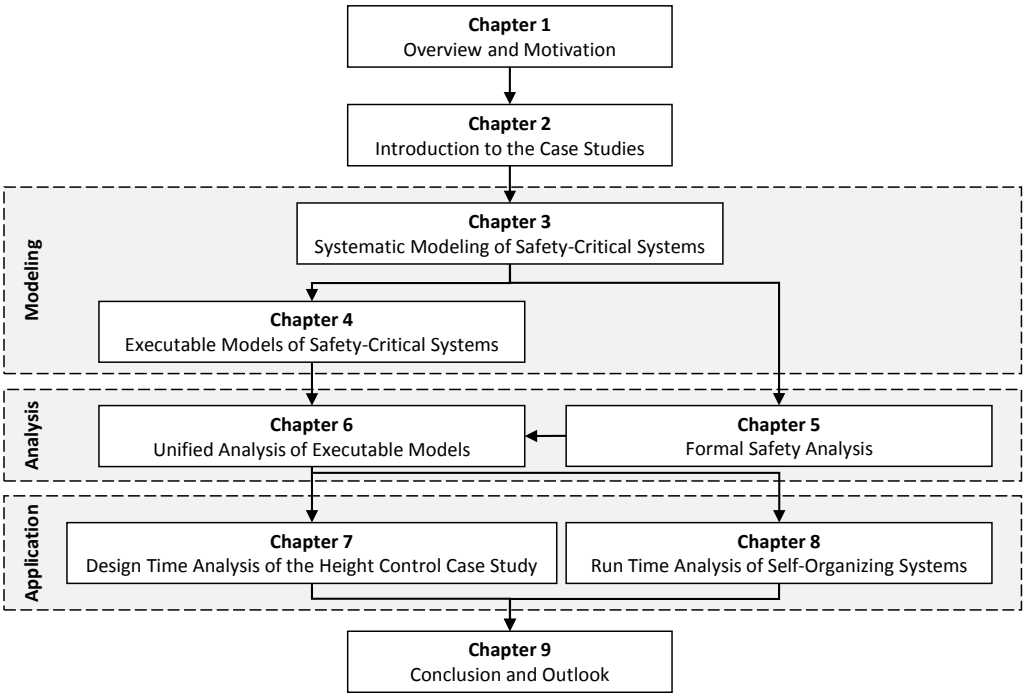


Figure 1.5. An overview of the thesis' general structure and the dependencies between the individual chapters. In the Modeling part, a systematic modeling approach for safety-critical systems is presented that is subsequently used to create executable system models. The modeling concepts influence the Analysis part that provides the formal foundations for model-based safety analysis and bridges the gap between the executable models and the formal analysis techniques. The Application part applies the modeling concepts and analysis techniques to some case studies, conducting the analyses either at design time or at run time.

specific in any way. Therefore, most of the contributions could be transferred to other safety analysis tools and techniques as well, many of which are discussed in more detail in the related work sections throughout this thesis. In general, the S# framework stands out from these other tools due to its unique unified model execution approach: S# combines a high-level, executable modeling language rooted in a systematic modeling approach with various analyses in the form of simulations, visualizations, and efficient, formal safety analyses; the efficiency of the latter can even be further improved through the use of model-specific heuristics.

The **Introduction to the Case Studies** (Chapter 2) describes several different safety-critical systems that are modeled and analyzed throughout the thesis using S#. Except for the simple running example, the case studies are inspired by industrial applications coming from the railway, traffic control, and medical domains as well as the domain of self-organizing resource-flow systems. Each case study presents a different challenge for formal safety analysis approaches that are subsequently tackled and discussed in the remaining chapters.

Systematic Modeling of Safety-Critical Systems (Chapter 3) fosters the creation of high quality models to improve model comprehensibility as well as model adequacy.

A SysML-inspired, component-oriented, systematic modeling approach is introduced based on the control-theoretical core concept of feedback loops between controllers, sensors, plants, and actuators of safety-critical systems. The approach is amended with systematic fault modeling concepts based on well-known notions and techniques such as fault activation or fault injection. Without any crucial restrictions concerning modeling flexibility, the approach fosters the creation of adequate models by preventing a certain class of fault modeling mistakes; namely, non-conservative fault injection.

The S# framework facilitates the systematic creation of **Executable Models of Safety-Critical Systems** (Chapter 4) based on the C# programming language and the .NET run time environment [105, 110]. It allows for modular and flexible model composition, also supporting modeling and analysis of multiple system design variants. First-class tool support results from not inventing a completely new modeling language; instead, S# provides a domain specific language embedded into C# that inherits the tooling available for regular .NET-based software development. When modeling safety-critical systems with S#, standard .NET development tools and libraries can therefore be used, including, among others, debuggers, version control software such as Git, continuous integration with regression tests on build servers, as well as code editing features such as refactorings and code completion.

Fault-aware Kripke structures and fault-aware Linear Temporal Logic are introduced as the foundations of **Formal Safety Analysis** (Chapter 5), making explicit-state model checking of safety-critical systems feasible: Analysis efficiency is increased by up to three orders of magnitude by enabling the model checker to ignore faults in situations they are irrelevant in. Moreover, the notion of critical fault sets is redefined to allow DCCA, the model checking-based formal safety analysis technique used and improved throughout this thesis, to be conducted using Linear Temporal Logic in addition to Computation Tree Logic. More importantly, this change also results in witnesses being generated by the model checker that show how critical fault sets can indeed cause a hazard; previously, witnesses showed how non-critical fault sets could not result in a hazard, which is less useful in practice. Furthermore, the notion of fault injection is formally specified in order to prevent unintended non-conservative fault injections and its dual, fault removal, is defined as well; DCCA can make use of the latter to more efficiently identify minimal critical fault sets. All of these results are not S#-specific in any way and are therefore also applicable to other safety analysis tools.

For **Unified Analysis of Executable Models** (Chapter 6), the S# framework conceptually generates fault-aware Kripke structures on-the-fly for analysis by an explicit-state model checker. Thus, Chapter 6 applies the formal foundations from Chapter 5 to S#, also enhancing DCCA with heuristics that further try to reduce analysis times. Due to the executability of its models, the S# framework also supports model simulation and visualization for non-exhaustive model exploration, debugging, and visual inspection of models of safety-critical systems. S# therefore has an advantage over other tools and approaches for formal safety analysis by tightly integrating the development, debugging, and simulation of models with their formal analysis. Additionally, S#'s unified model execution approach guarantees semantic consistency between model simulations, model visualizations, and explicit-state model checking.

A **Design Time Analysis of the Height Control Case Study** (Chapter 7) with S# and DCCA evaluates and demonstrates the applicability of the tools and techniques developed in this thesis. Particular emphasis is placed on the modeling of different design variants of the height control, with variants of the model being composed together using the reflection capabilities provided by the .NET framework underlying S#.

For **Run Time Analysis of Self-Organizing Systems** (Chapter 8), the analyzed models must be decomposed depending on which faults can be tolerated by the systems' self-organization mechanisms. Moreover, S#'s flexible model composition capabilities are taken advantage of to enable virtual commissioning of self-organizing systems, i.e., safety analyses are conducted while the systems are already in operation. To cope with the large state spaces of models of self-organizing systems as well as their typically high levels of redundancy, DCCA efficiency is improved through model-specific strategies and heuristics that are designed to find minimal critical fault sets significantly faster in many cases.

Conclusion and Outlook (Chapter 9) summarize the contributions of this thesis as well as the lessons learned, also giving an outlook to future work.

Summary. Six safety-critical systems from a range of different application domains are introduced in this chapter. Except for the simple running example that is primarily used to introduce and motivate the modeling and analysis concepts and improvements of this thesis, each of the other case studies presents a different challenge for model-based safety analysis. Two of the case studies are only briefly described as they are only used to evaluate the overall modeling and analysis approach but are otherwise not discussed in greater detail.

Publications. The case studies are partially modeled and analyzed in [81, 82, 84, 120, 130]; the complete S# models are available in the S# repository [101]. This introduction is based on the case study descriptions from the S# Wiki [102].

2 Introduction to the Case Studies

This chapter gives an overview of the six safety-critical systems serving as the case studies throughout this thesis: A controller regulating the pressure within a pressure tank, the height control system of the Elbe Tunnel in Hamburg, Germany, a self-organizing robot cell, self-organized production of personalized medicine, a radio-controlled railroad crossing, and a hemodialysis machine. The pressure tank case study is used as the primary running example as it is both small and easy to understand, yet sufficient to illustrate and motivate the basic concepts of the modeling and analysis techniques. The height control is a more complex case study, highlighting the flexible model composition capabilities of the modeling approach as well as the efficiency improvements of the formal analysis techniques introduced in the subsequent chapters. The two self-organizing systems emphasize the need for more flexible, adaptable, and configurable modeling capabilities to instantiate their numerous system configurations. Additionally, they also require significant analysis improvements in the form of heuristics to cope with the large number of relevant faults and different configurations that have to be analyzed.

The first four of the aforementioned case studies are primarily used to illustrate the main contributions of this thesis, while the railroad crossing and the hemodialysis machine, on the other hand, are not discussed in full detail in the remainder. Nevertheless, they are used to motivate and explain some of the safety analysis concepts and they provide additional data points for the evaluations of analysis efficiency. Furthermore, they help to demonstrate the broad applicability of the modeling and analysis techniques to a wide range of different domains.

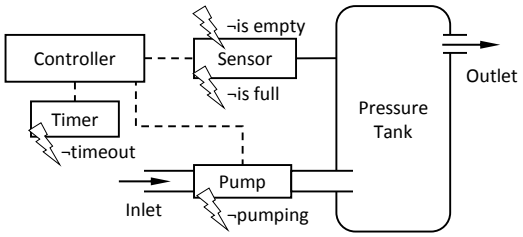


Figure 2.1. A schematic overview of the components constituting the pressure tank case study: The software controller uses the pressure sensor and the hardware timer to decide whether the pump should fill the tank with a fluid. The considered faults are indicated by lightning bolts near the affected components; the sensor's fault ¬is full, for instance, indicates a suppression of the sensor's notification about a full tank.

2.1 Pressure Tank

The pressure tank case study is a very simple safety-critical system that is used to motivate and explain the concepts of S#'s modeling and analysis approach. In order to make it as simple as possible while still retaining the spirit of the original version of the case study [193], the electronic control system of the original version is replaced by a software-based controller implementing the same functionality. A schematic overview of the simplified case study is shown in Figure 2.1: The fluid contained in the pressure tank is refilled by the pump at the tank's inlet; at the outlet, the fluid can be removed. The pump is activated and deactivated by a software controller. The pressure sensor signals the controller when the pressure limit is reached or when the tank is empty, causing the controller to deactivate or activate the pump, respectively. How and why fluid is removed from the tank is not considered in detail and thus constant fluid removal is simply assumed. Once the tank becomes empty, a new refill cycle begins. The system is safety-critical because of the hazard of tank ruptures that might injure people standing nearby. To tolerate pressure sensor faults, the controller disables the pump after 60 seconds of continuous operation as it would risk a tank rupture otherwise. For time measurements, the controller uses the hardware timer.

Faults. Only the four faults shown in Figure 2.1 are considered: The first two faults are suppression faults of the pressure sensor, causing it to no longer report that either the pressure limit is reached (¬is full) or that the tank is empty (¬is empty). The third fault prevents the timer from reporting a timeout (¬timeout) and the fourth one results in a failure of the pump, preventing it from filling the tank (¬pumping).

Hazards. The main hazard are tank ruptures due to overpressure that might injure nearby people. A completely empty tank is another hazard as it might cause problems for the consumer of the fluid stored in the tank. For such a simple system, it is obvious that a tank rupture can only occur when both the sensor's ¬is full fault and the timer's ¬timeout fault occur. The hazard of a completely depleted tank, by contrast, is caused either by ¬pumping or by ¬is empty. For more complex systems with a wider variety of faults, however, informal safety analyses typically cannot be carried out as easily. Instead, model-based safety analysis techniques must be used to compute the relationships between hazards and fault both automatically and thoroughly.

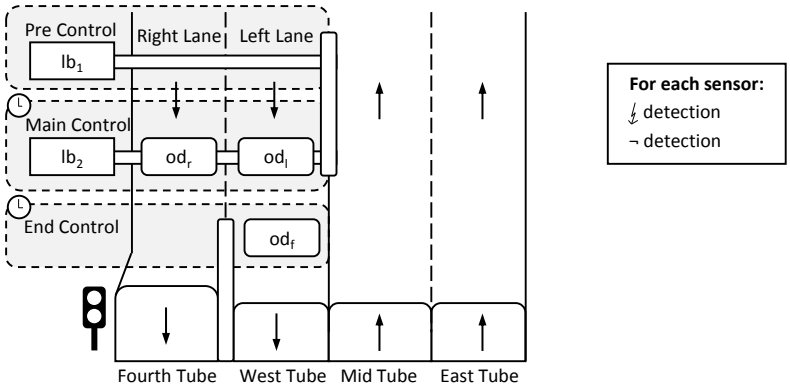


Figure 2.2. A schematic overview of the height control case study: There are four lanes entering and exiting the tunnel's four tubes at the bottom, with the arrows indicating the driving direction. Overheight vehicles are only allowed to enter the fourth tube driving downwards. The height control system consists of a pre, main, and end control and various light barriers and overhead detectors to check for overheight vehicles on the wrong lane. The considered faults are the same for each of the five sensors: ¬detection prevents the sensor from detecting a vehicle, while ↯ detection causes the sensor to spuriously report a vehicle that is not there.

2.2 Height Control System

A height control system was added to the Elbe Tunnel in Hamburg, Germany, when the fourth tube was built in 2002 [158]. Overheight vehicles are only allowed to use the new tube; otherwise, they might collide with the ceiling of the tunnel. The height control is therefore responsible for closing the tunnel when it detects an overheight vehicle that tries to enter one of the old tubes. Figure 2.2 shows a schematic overview of the system and the road layout. Only the two lanes from top to bottom are of interest, as the tunnel is simply closed whenever an overheight vehicle is detected driving into the other direction. Overall, the height control consists of five sensors: Two light barriers lb_1 and lb_2 as well as three overhead detectors od_r, od_l, and od_f. The sensors are grouped into the pre, main, and end controls, with the latter two also making use of a timer.

The light barriers span the entire width of both lanes, whereas each overhead detector is positioned hovering only one of the lanes. Consequently, the light barriers can only report that an overheight vehicle passes by, but cannot distinguish between overheight vehicles on the left lane or on the right lane; it is physically impossible to install the light barriers in such a way that they observe a single lane only. The overhead detectors, by contrast, can in fact distinguish between overheight vehicles on either of the lanes, but they cannot differentiate between overheight vehicles and regular non-overheight trucks. They are, however, not triggered by passenger cars. By contrast, the light barriers are positioned high enough to ensure that they are only triggered by passing overheight vehicles. The height control therefore has to combine the data of both types of sensors to determine the positions of overheight vehicles in the observed area.

When no overheight vehicles are approaching the tunnel, only the pre control is active, i.e., the sensors of the main and end controls are deactivated. When lb_1 detects an overheight vehicle, the main control is activated, enabling its sensors and starting its

timer. Also, an internal counter is increased that counts the number of overheight vehicles within the main control area. The main control is deactivated when a vehicle is reported by lb_2 and od_r or od_l and the counter reaches zero, or the main control's timer times out. If the main control discovers an overheight vehicle on the right lane, the end control enables its sensor and starts its timer; if, on the other hand, an overheight vehicle is detected on the left lane, the tunnel is closed immediately even though the vehicle is still able to switch to the right lane. When the end control does not detect an overheight vehicle before its timer runs out, it is deactivated; otherwise, the tunnel is closed. The road layout makes it impossible for any vehicle to switch lanes after it has passed od_f, but between the main and end controls, vehicles are free to do so.

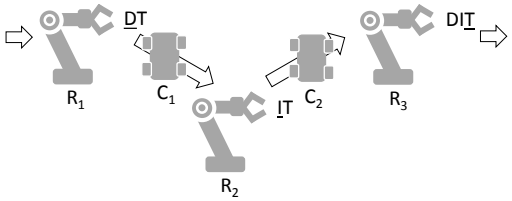
Faults. There are two kinds of faults for each of the five sensors: Misdetections ($\neg$detection) are false negatives, that is, a sensor does not report a vehicle passing by that it should detect. False detections ($\lightning$ detection), on the contrary, are false positives, so a sensor detects something that is not a vehicle, but, say, a bird.

Hazards. The two analyzed hazards conflict with each other: On the one hand, the height control system is designed to prevent collisions by closing the tunnel whenever an overheight vehicle is about to enter the wrong tube. On the other hand, the height control is also designed to prevent false alarms for economic reasons, as unnecessary closures cause traffic jams and economical losses. The system design takes both hazards into account, trying to strike a balance between them. In the subsequent chapters, safety analyses are conducted for both hazards, but techniques for safety optimizations that balance both hazards are out of scope; for example, Güdemann and Ortmeier [68] discuss multi-objective optimization of antagonistic safety goals in more detail. Previous analyses [158, 159] of the case study revealed that the height control system as introduced above is functionally incorrect due to a design flaw, that is, collisions and false alarms can happen without the occurrence of any hardware faults. Design alternatives were proposed that add additional sensors in order to fix the problem [158], while potentially introducing other safety issues at the same time as well.

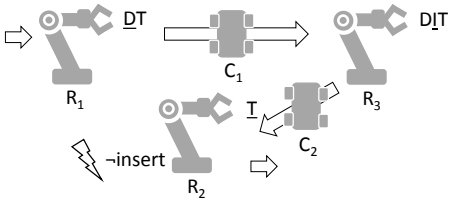
Challenges. Any changes to the design of a system necessitate additional safety analyses. Consequently, the case study's first challenge is convenient support for variant modeling, preferably also allowing for combinations of different orthogonal design variants in order to analyze the safety of the different design alternatives that were proposed. Additionally, the large number of faults for some of the design variants and the nondeterminism that the vehicles introduce into the model make formal safety analyses expensive. Therefore, the second challenge is to make the models more modular and structured in order to increase comprehensibility and to facilitate variant modeling without sacrificing analysis efficiency at the same time.

2.3 Self-Organizing Robot Cell

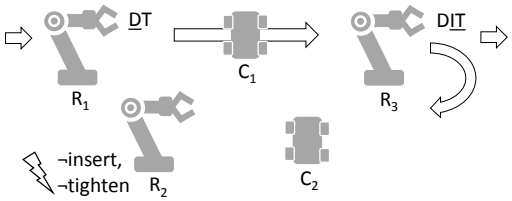
The self-organizing robot cell [81] consists of robots and carts. The carts transport workpieces between the robots that have several switchable tools such as drills and screwdrivers. A production task requires a workpiece to be processed by a sequence of tool applications; for the case study, the required processing sequence is to drill a hole,



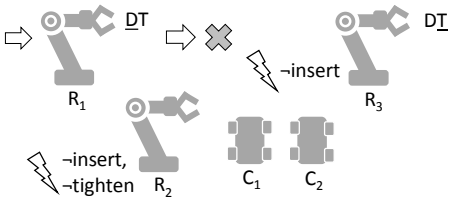
(a) The resource flow starts to the left where R_1 drills a hole into incoming workpieces. Workpieces are successively transported by C_1 and C_2 to R_2 and R_3, applying their insert and tighten tools, respectively. Once R_3 is done, the workpieces leave the system.



(b) After R_2 loses its insert tool, the resource flow is reconfigured: R_2 and R_3 switch roles, each taking over the previous role of the other. The carts transport the workpieces to R_3 before R_2 to guarantee the correct processing order.



(c) After a failure of R_2's tighten tool, R_2 and C_2 are no longer used at all, as R_2 has no more working tools available. Instead, R_3 uses two of its tools on the workpieces successively: Insert, then tighten.



(d) After R_3's insert tool is lost too, the robot cell can no longer be reconfigured such that the task can be completed: There is no robot with an insert tool that could follow R_1's drilling.

Figure 2.3. A schematic overview of the self-organizing robot cell case study with three robots R_1, R_2, and R_3 as well as two carts C_1 and C_2 establishing the resource flow between them. The task is to apply the drill, insert, and tighten capabilities to all incoming workpieces. Each robot's available tools are shown to its right, with D, I, and T denoting the drill, insert, and tighten tools, respectively; the currently allocated ones are underlined. Figure 2.3a shows an exemplary configuration of the robot cell. As depicted in Figures 2.3b to 2.3d, faults result in tool losses that self-organization can cope with by reconfiguring the resource flow; eventually, however, no further reconfiguration is possible as the system runs out of redundancy.

insert a screw, then tighten the screw. Each robot has roles assigned that indicate which tools it has to apply on the workpieces and in which order, while the carts know the robots they have to transport the workpieces between. The robot cell is self-organizing as it can adapt its structure and behavior by itself to compensate for broken tools or to incorporate new tools, robots, or carts, for instance. That is, these events result in reconfigurations of the transportation routes and tools applied by the robots as shown in Figure 2.3 in order to ensure correct workpiece processing.

Faults. The system's self-organization capabilities cannot cope with all faults that occur during the lifetime of the system: Some faults are outside of its reach, either because their occurrences cannot be detected or there is no appropriate reaction to their occurrence. For example, carts might move to the wrong robot or robots and carts might not receive reconfiguration information due to, for instance, networking problems. On the other hand, there are faults that self-organization is designed to handle such as broken tools, robots that are no longer working at all, or carts whose assigned paths are blocked. At some point, however, there is not enough redundancy left in the system

to continue operating safely after the occurrence of enough faults, in which case no further configurations exist. In the case study, for instance, no more configurations can be found as soon as all tools of the same kind no longer work.

Hazards. There are two hazards that are most relevant for the case study: The first one occurs when a workpiece is damaged, i.e., processed in an incorrect order. The second hazard occurs when some workpiece is never finished, i.e., not processed any further. In addition to these system-level hazards, it is also of interest to determine all combinations of faults that can be compensated through reconfigurations. Safety analyses can be conducted that show the limits of self-organization, computing the sets of faults whose occurrences make any further reconfiguration impossible.

Challenges. The robot cell is a complex case study that dynamically adapts its behavior and structure to changes in its environment and occurrences of faults, resulting in system configurations that cannot be predicted at design time. It is therefore necessary to postpone model-based safety analyses to run time when the actual configurations of the system are known. However, the high level of redundancy resulting from the case study's self-organization capabilities is problematic for the efficiency of state-of-the-art model-based safety analyses techniques: The models have large state spaces both because of the high number of constituent components as well as the large number of potential faults. Additionally, the latter results in a combinatorial explosion of the possible fault sets that may lead to hazards. Therefore, techniques must be developed in order to modularize safety analyses such that the exponential complexity can be reduced. Additionally, the functional correctness of the self-organization mechanism must be established, for instance by systematic testing.

2.4 Self-Organized Production of Personalized Medicine

The self-organizing production system for personalized medicine [37] adapts dosage and composition of medical ingredients to a specific individual. Personalized medicine improves treatment efficiency by avoiding costly and potentially dangerous trial and error processes to find the right kind of medicine for a patient [167]. The case study consists of several production stations that are connected by conveyor belts as illustrated by Figure 2.4. There are three kinds of stations: Stations that load pill containers on a conveyor belt, stations that dispense different ingredients, and stations that remove processed containers and palletize them. Additionally, there are three types of ingredients that are depicted in different shades of gray in Figure 2.4. Recipes specify how the pill containers should be filled through ordered applications of specific ingredient types and amounts. These recipes are unknown beforehand as they enter the system during run time, for example via web services whenever a personalized product is ordered. For each recipe, the system's reconfiguration mechanism finds a sequence of neighboring stations that is capable of processing pill containers according to the recipe. In contrast to the robot cell case study, the conveyor belts establish the resource flow statically, i.e., conveyor belts cannot be dynamically rerouted similar to the carts.

Faults. Each station can fail completely ($\neg$response), representing the total shutdown of the station due to power loss, a crash in the station's controller, or similar reasons.

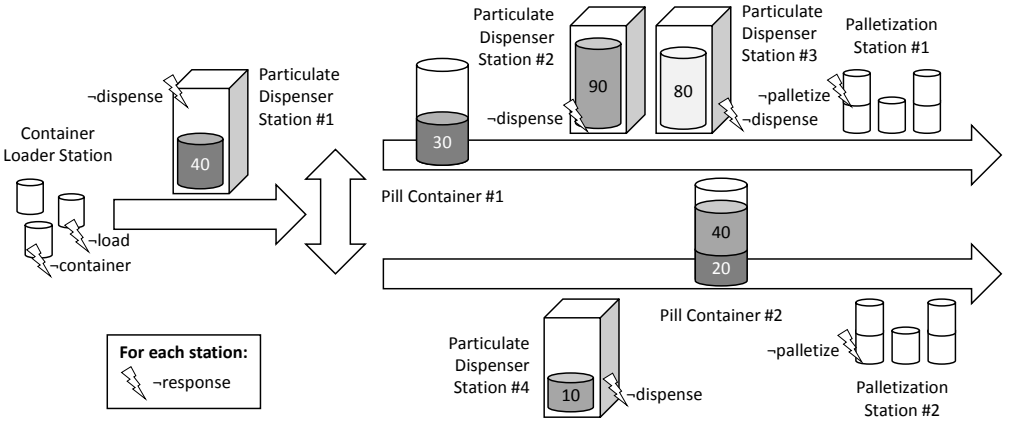


Figure 2.4. A schematic overview of the self-organizing production system case study for personalized medicine. The stations are connected by conveyor belts; they either load pill containers onto the belts, palletize them, or dispense specific amounts of one or more ingredients into the containers. The overview shows a very simple configuration of the production cell: The pill containers can take two different routes through the system, with only the upper route being able to dispense all three ingredient types into the containers. Which routes the pill containers take therefore depends on their recipes and the amount of ingredients left. For example, the lower route has only 10 units of the medium gray ingredient left, causing all pill containers requiring more than 10 units to be routed to the upper section of the production cell.

This fault disables any processing capabilities of the station and also cuts all resource-flow paths running through it, that is, no pill containers on the station's associated conveyor belt can pass through. On the other hand, there are faults representing situations in which no pill containers are left (¬container), the container loading or palletization capabilities of a station fails (¬load or ¬palletize, respectively), or the tank of some ingredient type is no longer able to dispense the ingredient (¬dispense for some ingredient type). All of these faults only affect the corresponding station's respective capabilities, not the resource-flow paths.

Hazard. The case study is only used to determine the limits of the system's reconfiguration mechanism, i.e., situations in which the system is incapable of finding any further configurations. Reconfiguration becomes impossible once fault occurrences used up all of the system's redundancy, for instance. Additionally, the system is unable to continue processing whenever it runs out of ingredients, which is to be expected and must consequently be considered during operation and reconfiguration.

Challenges. To determine the limits of the case study's self-organization mechanism, large numbers of faults must be checked for their potential to prevent the system from further reconfiguration. Due to the inherent redundancy, the number of faults that are required to occur before no new configurations can be found is typically rather large. For example, if there are n stations administering the light gray particulate and all of them are connected together, all n light gray particulate dispensers must fail to prevent further reconfiguration. However, model checking-based safety analysis scales exponentially with the number of faults to be analyzed [83], therefore requiring the development of heuristics to lessen the combinatorial explosion.

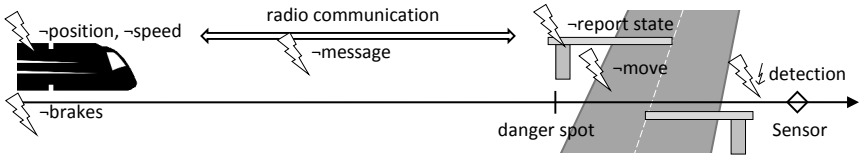


Figure 2.5. A schematic overview of the radio-controlled railroad crossing case study: The train approaches the crossing from the left, communicating with the crossing via radio to initiate its closing procedure. The considered faults are indicated by lightning bolts near the affected components; for instance, ¬move indicates the barrier motor to be broken such that the barriers no longer open or close. By contrast, ↯ detection indicates that the sensor reports a detection of a train even though no train has passed the sensor.

2.5 Radio-Controlled Railroad Crossing

The radio-controlled railroad crossing case study [53, 93] proposes a mechanism to control railroad crossings in a decentralized way. Many of the sensors and signals traditionally placed alongside the train tracks are replaced by radio communication and software computations to make the overall system more robust, scalable, and cost effective. A control software on-board the trains knows about the locations of the crossings that lie ahead and secures them just in time via wireless, radio-based communication. Among other problems, the controller takes lost communication messages and malfunctions of the crossing into account. It initiates an emergency stop right before reaching the next crossing on the train's way if the crossing is not known to be in a secured state with the barriers closed. Figure 2.5 shows a train approaching a crossing; it sends a message to the crossing requesting it to initiate its closing procedure. This procedure takes some time as the traffic lights must first be activated before the barriers are allowed to begin closing, for instance. Once the barriers are closed, the crossing is secured and the train is allowed to pass the danger spot. During the time the barriers are closing, the train continues to approach the crossing. After the crossing has had enough time to secure itself, the train's control software queries the crossing's state via radio. If everything goes well, the crossing responds with a message indicating that it is secured and that the train can safely pass. If, however, the train does not receive the message in time, it initiates an emergency stop. The sensor behind the crossing signals the crossing to open once the approaching train passes by.

Faults. Seven faults are depicted in Figure 2.5: The train's odometer might report incorrect position or speed values (¬position and ¬speed, respectively). The train's brakes might malfunction, preventing the train from stopping (¬brakes). The sensor checking whether the barriers are closed might incorrectly report that non-closed barriers are closed or vice versa (¬report state) or the barrier motor might get stuck, preventing the barriers from opening or closing (¬move). Lastly, the sensor that causes the crossing to be opened again might incorrectly report that the train has passed (↯ detection) or communication messages transmitted via radio might get lost (¬message).

Hazards. Collision avoidance is the system's primary safety goal, i.e., the system is designed to avoid the hazard of a train passing an unsecured crossing. Another hazard is that of the erratic human behavior where drivers circumvent the closed barriers; it

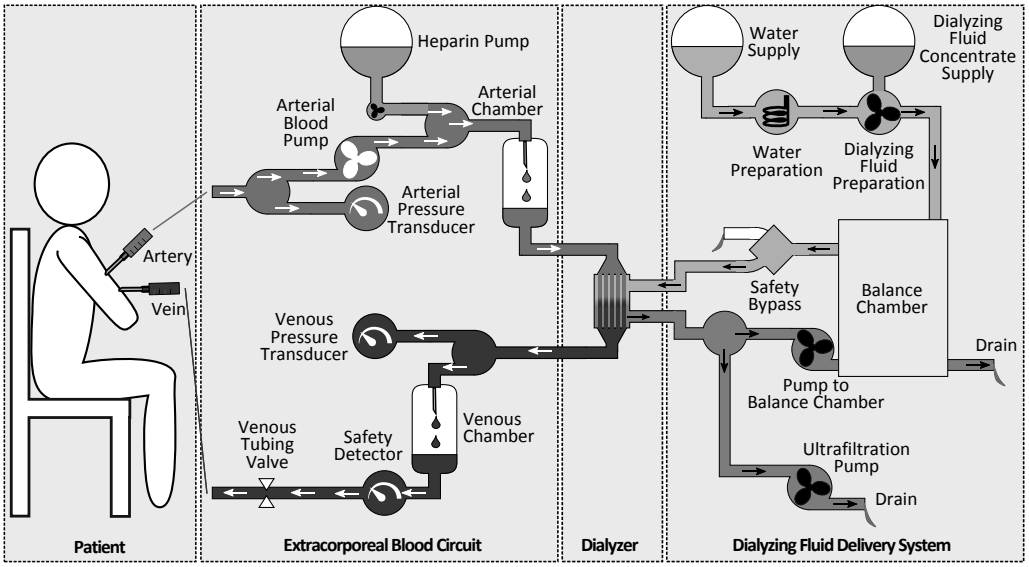


Figure 2.6. Overview of a hemodialysis machine connected to a patient. The extracorporeal blood circuit pumps the blood through the dialyzer, where it is cleaned and subsequently pumped back into the patient. The dialyzer uses an ultrafiltration process to remove waste products from the blood with the dialyzing fluid delivery system being responsible for providing the necessary dialyzing fluid to do so.

occurs if the barriers remain closed for much longer than necessary. These two hazards are antagonistic, as the former can be avoided by never opening the barriers while the latter is made impossible by never closing them. Therefore, a tradeoff must be found that balances both hazards, a problem which has already been discussed in prior work [68, 80]. In the remainder, the discussion is therefore restricted to the first hazard, the hazard of collisions.

Challenges. The challenge primarily lies in the systematic creation of a comprehensible model that reflects the fact that the two physically separate controllers for the train and the barriers are connected together via a radio-based communication channel. From an analysis point of view, the case study can have a large state space depending on the necessary discretizations of the train's position and speed. However, only a small percentage of the states are actually reachable.

2.6 Hemodialysis Machine

The human body creates metabolic waste products that the kidneys usually remove from the blood. When they fail, a hemodialysis machine must be used for this removal process instead [47, 140]. As such a machine directly influences the chemical composition of a patient's blood, it is safety-critical. A conceptual overview of a hemodialysis machine connected to a patient is given by Figure 2.6. The patient's artery and vein are connected to the extracorporeal blood circuit through syringes in order to deliver the blood from the patient to the dialyzer and subsequently back again to the patient once it has

been cleaned of the metabolic waste products. The dialyzer cleans the blood with a semipermeable membrane through which waste products diffuse from the blood into the dialyzing fluid provided by the dialyzing fluid delivery system. A total of nine faults are considered in the case study that affect various components of all three parts of the system. As the case study is only used for analysis efficiency evaluations in the remainder, its inner workings and the faults are not explained in more detail; Leupolz et al. [130] give a detailed report on a S#-based analysis of the case study.

Hazards. The hemodialysis machine is used to cleanse a patient's blood from metabolic waste products. Consequently, a hazard occurs when the device fails to do so for various reasons after fault occurrences, that is, dialysis is unsuccessful when the overall dialyzing process is completed but the patient's blood is still not fully cleaned. Additionally, there is the hazard of contaminated blood entering the patient which the safety detector and the venous tubing valve shown in Figure 2.6 are designed to prevent. But like all safety measures, they can fail, resulting in potentially lethal doses of contaminated blood being pumped into the patient.

Challenges. The case study consists of a multitude of different components connected together through fluid flows. In order to facilitate formal safety analysis, these flows must be represented in a model, abstracting from the underlying complex physical laws in order to conduct qualitative safety analyses without taking all physical details into account. The main challenge therefore lies in a systematic and modular modeling approach to create a comprehensive model that allows for formal safety analyses.

Summary and Outlook. The challenges provided by the aforementioned case studies fall into three main categories: Creating adequate models, increasing safety analysis efficiency, and coping with self-organization. The remainder of this thesis discusses several approaches to tackle these challenges, applying them to and evaluating them with the height control case study in Chapter 7 as well as the two self-organization case studies in Chapter 8; the pressure tank case study is the running example used throughout Chapters 3 to 6. A systematic modeling approach for safety-critical systems is introduced in Chapter 3 and supported by the S# modeling language introduced in Chapter 4. Subsequently, an efficient safety analysis technique based on model checking is introduced in Chapter 5 and incorporated into the S# framework in Chapter 6. An approach to tackle the unique challenges of self-organizing systems by deferring safety analyses to run time is discussed in Chapter 8.

Summary. The systematic modeling approach for safety-critical systems introduced in the following supports the creation of adequate models, a prerequisite for model-based safety analysis. Taking inspirations from SysML modeling concepts as well as control theory [36, 62, 131, 153], structural and behavioral models are created for the analyzed systems, emphasizing the feedback loops between plants and controller components. In a separate step, faults are incorporated into the models using the fault terminology and error propagation concepts summarized by Avižienis et al. [9].

Publications. This chapter is an extended version of the work published in [84].

3

Systematic Modeling of Safety-Critical Systems

This chapter introduces and motivates a systematic modeling approach for safety-critical systems that supports the creation of adequate models for formal safety analysis. Systematic modeling takes advantage of basic concepts from systems and control theory as well as embedded systems development, introduced in Section 3.1. Based on these concepts, structural and behavioral modeling guidelines are discussed in Sections 3.2 and 3.3. Subsequently, Section 3.4 introduces a systematic fault modeling approach based on well-known fault-related concepts such as fault injection and fault activation. This chapter exclusively uses SysML [153] for systems modeling. SysML is a semi-formal systems modeling language based on UML [154] that is well-established in the field of software engineering and embedded systems development, including safety-critical systems [79]. SysML defines many different types of models, all of which are tailored towards describing certain structural or behavioral aspects of a system. For instance,

sequence diagrams can be used to specify the dynamic behavior of a system while block definition diagrams depict the different parts a system can be composed of. Overall, combinations of different SysML model types allow for complete and comprehensive descriptions of safety-critical systems.

The modeling guidelines presented in this chapter are derived from and strongly influenced by the control-theoretical view of safety-critical systems as discussed by Leveson [131], highlighting the importance of the correct representation of controllers, sensors, actuators, and plants within the models for all safety-related modeling and analysis activities. This partitioning is one of the most fundamental aspects of modeling safety-critical systems and is therefore imperative to get right, especially once faults are injected into the models in order to facilitate formal safety analyses. The guidelines for fault modeling are extensions of the work by Ortmeier et al. [160], systematizing the experiences of working on the case studies from Chapter 2. On the other hand, some of the guidelines are derived from general best practices and basic principles of software engineering that cope with system complexity and increase modularity and comprehensibility; e.g., separation of concerns, single responsibility, low coupling, and others [33] apply just as well when modeling safety-critical systems.

There are both analytical and constructive techniques and guidelines for building safety-critical systems: The former assess system safety, identifying unacceptable safety issues that have to be resolved before the system can be built or deployed. The latter, by contrast, consist of constructive measures for designing and building safety-critical systems. While both kinds of techniques are important during system development, this thesis focuses exclusively on analytical activities. Thus, the systematic modeling approach provides guidelines on how the models of safety-critical systems should be structured, how behavior should be described, and how faults should be injected into the models to allow for formal safety analyses. These guidelines, however, are not concerned with the potential ways to actually architect and design safety-critical systems, nor do they consider how relevant faults can be identified. Both of these tasks fall into the domain of constructive safety engineering for which, non-surprisingly, various techniques exist. For example, there are multiple architectural safety patterns that describe well-established and proven techniques to design safety-critical functionality of a system [6, 87, 172] such as triple modular redundancy that is briefly discussed in Section 3.2.3. For fault identification, classical safety analysis techniques such as FMEA [166] can be used, while more formal approaches such as failure-sensitive specification [155] are available as well. Additionally, programming libraries, e.g., Uncertain<T> [23] for working with unreliable sensor data, help to avoid common software bugs, potentially preventing development faults that might lead to hazards during system operation.

3.1 System and Model Basics

Most safety-critical systems belong to the class of reactive systems [204]: They constantly and continuously interact with their surroundings to enforce reliable and safe behavior whenever possible. The systems respond to environmental stimuli depending on their current states, causing internal state changes as well as effects on their envi-

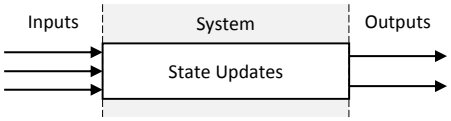


Figure 3.1. Overview of the basic structure of a system: System inputs trigger internal state changes, usually depending on the system's previous state. The outputs represent the responses to the inputs which are also typically influenced by the system's state.

ronments. Safety-critical systems only have indirect means to monitor and influence their environments, so they typically cannot enforce or prevent certain environment behavior under all circumstances, particularly when their abilities are limited by one or more faults. In general, safety-critical systems employ a form of control over their environment. Control is a concept originating from systems theory as well as control theory, which study systems in general and behavior modifications of dynamic systems using feedback in particular [36, 62, 131]. The following subsections briefly discuss some system and control basics that are subsequently applied to safety-critical systems in order to motivate most aspects of the systematic modeling approach.

3.1.1 System Models

A system is a combination of different elements interacting with each other in order to achieve the functions that the system is intended to perform [106]. Due to this open definition of the notion of a system, systems theory is applicable to all sorts of physical and non-physical objects, including technical hardware- and software-based controllers, software protocols, natural laws, human behavior, or economic mechanisms. The distinguishing factor between systems and models is that systems are real, whereas models are abstractions approximating the systems in accordance with the system aspects the models are intended to describe. In general, models can be created for all kinds of systems; the opposite, however, is not necessarily true: Contradictions within a model or absurd physical assumptions make it impossible to implement the model as there can be no real system corresponding to it. For instance, a model of the railroad crossing case study where the train moves faster than the speed of light is certainly conceivable, yet a corresponding system cannot be implemented (at the moment) due to the model bending the known laws of physics.

A formal model is a mathematical description of a system that allows for rigorous reasoning about the system's behavior. Typically, there are multiple approaches to mathematically express the system's behavior, some of which are more appropriate for the task at hand, such as formal safety analysis, than others. Hence, modeling requires creativity and cannot be automated in a general way. However, a model always consists of the three main parts shown in Figure 3.1 that correspond to the system's inputs, its internal state updates, as well as its outputs.

Input & Output. In order for a model to be of any value, it must describe the system's responses to certain relevant stimuli. Systems generally have a set of inputs that provoke certain outputs, i.e., the system responds to its input stimuli by producing some output. For example, a model of the pressure tank might have two inputs: The amount of fluid

that is pumped into it and the amount of fluid that is removed from it. The tank's output could be modeled as a function that indicates whether the tank is full, empty, anything in between, or ruptured. In general, however, system outputs cannot be defined over the inputs alone; they also depend on the system's state as discussed in the following.

State Vector. The system's state vector contains all the information required to determine the system's outputs based on its state as well as the inputs. Due to their inherently nondeterministic nature, safety-critical systems generally have multiple possible outputs for the same states and inputs; faults, in particular, can affect the system's behavior at unforeseeable times, therefore making the outputs unpredictable. However, the state vector must allow for predicting all future outputs given some inputs, hence faults must be contained in the model, for example as additional, nondeterministic system inputs. For the pressure tank example, the state vector might consist of the tank's current pressure level as well as a flag indicating whether the tank is ruptured. The latter must be part of the state vector as the system's response to some stimuli obviously depends on whether the tank is already ruptured; without the knowledge of whether the tank has already ruptured, future outputs could not be predicted.

State Equations. A system's state space consists of all possible system states over time, that is, the state space is induced by the state vector. The state space is completely specified by giving some initial states of the system as well as state equations describing how the state changes over time in response to the inputs. For continuous-state, continuous-time systems, state equations amount to differential equations describing the overall state vector evolution. Discrete-state systems, on the other hand, generally use discrete functions to describe the possible state changes, i.e., difference equations.

Discrete-State, Discrete-Time Models. In the pressure tank example, the part of the state vector representing the pressure level is a subset of the real numbers. The tank's rupture flag, on the other hand, is obviously modeled more intuitively as a Boolean value. The overall state space of the pressure tank model is therefore a hybrid of continuous and discrete state information. This thesis, however, makes the simplifying assumption that all state information is modeled in a discrete way. Similarly, the assumption of zero execution time introduced in Section 3.1.3 approximates time by a sequence of discrete time points $t_0 < t_1 < \ldots$ with $t_{i+1} - t_i = \Delta t$ for all i, resulting in discrete-time models where the same amount of time passes between all consecutive points in time. For digital systems such as software-based controllers, the above simplifications are inherently justified as continuous values or continuous time do not exist in the digital space. Most controllers of the case studies introduced in Chapter 2 are primarily software-based, for instance, whereas the plants, on the other hand, are often continuous in reality.

Discretization. Continuous state such as the pressure tank's pressure level or the train's position and speed in the railroad crossing case study can be discretized using, for example, standard numerical procedures for solving ordinary differential equations [34]. One of the most basic methods is the Euler method: To approximate a solution for a differential equation of the form $\dot{y}(t) = f(t, y(t))$ with initial value $y(t_0) = y_0$, the value y_{k+1} at time step t_{k+1} is approximated by $y_{k+1} = y_k + \Delta t \cdot f(t_k, y_k)$, where Δt denotes the step size and $t_k = t_0 + k \cdot \Delta t$. For example, the differential equation describing the train's position in the railroad crossing case study is $x(t) = \frac{1}{2}\ddot{x}(t) \cdot t^2 + \dot{x}(t) \cdot t$. Assuming

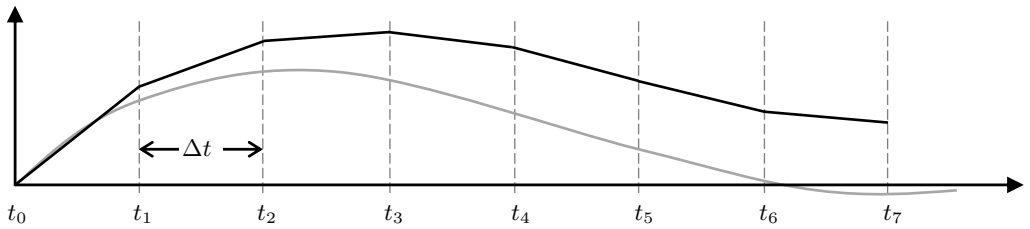


Figure 3.2. Illustration of the Euler method's approximation error. The continuous function drawn in gray is approximated with a step size of Δt by the discrete function drawn in black, the error increasing with each time step.

a constant acceleration $a = \ddot{x}(t)$, the train's position p and speed s can be approximated by $p_{k+1} = p_k + s_k \cdot \Delta t$ and $s_{k+1} = s_k + a \cdot \Delta t$ for each analyzed point in time. The global error of the Euler method is approximately proportional to the step size Δt [34] and can therefore become quite significant over time as illustrated by Figure 3.2. Still, the Euler method is often sufficient for model checking-based analyses of safety-critical systems as the number of analyzed steps is typically rather small to reduce analysis times, keeping the deviation from the actual values at a tolerable level. If necessary, however, accuracy can be improved by using more precise, but also more complex numerical techniques such as the Runge-Kutta method or the predictor-corrector method [34].

3.1.2 Control and Feedback

The pressure tank case study can be considered to consist of two main subsystems that influence each other: The first one is the pressure tank containing the fluid that is added and removed as previously discussed; the second one is the software controller that observes the pressure level and controls the pump that fills the tank. As these two parts can be seen as systems again, the modeling process can often be decomposed along the hierarchy of systems and subsystems. The distinction between a controller subsystem and a subsystem that is being controlled is common, especially in the context of safety-critical systems [137, 204]. In control theory, the controlled subsystem is usually called the plant [36]. Because of their influence on the systematic modeling approach for safety-critical systems, the concepts of control and feedback between controllers and plants are briefly introduced in the following. A structural overview of the closed-loop systems that result from control and feedback is shown in Figure 3.3.

Control. A controller is tasked with the selection of a plant's inputs such that the plant exhibits desirable behavior and fulfills its intended functions: It executes a control algorithm that sets the plant's inputs in such a way that undesirable plant behavior is prevented. In the pressure tank case study, for example, the controller tries to prevent unacceptable plant behavior such as tank ruptures, that is, the overall system is considered to be safe with regard to the hazard of ruptures if the controller succeeds in preventing them. Without any control, ruptures are very likely to occur, as the pump could be continuously enabled even though no fluid is removed from the tank.

Feedback. Feedback allows a controller to react to unexpected disturbances such as faults, making corrections to the plant's inputs in order to stipulate intended behavior.

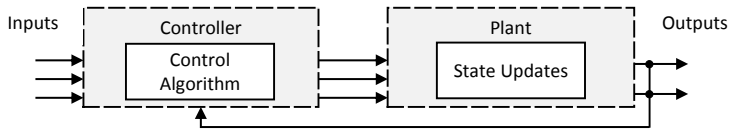


Figure 3.3. An overview of a closed-loop system with feedback: The controller modifies the plant's inputs to stipulate desired behavior. The plant's output is fed back into the controller, allowing it to react to unexpected disturbances such as faults. Similar to the plant, the controller could also have state (not shown in the figure) affecting its output, i.e., the inputs of the plant.

Typically, feedback means that a controller is able to observe parts of the plant's state in order to make more informed control decisions. Observing plant state usually requires additional sensors which increase development and system costs; in most real systems, the controller therefore only has access to a fraction of the plant's entire state vector. In the pressure tank case study, for example, the controller depends on the pressure sensor to provide feedback on whether the tank is full, influencing its decision to stop pumping. Without the sensor, the controller could only assume that the tank is full after a certain amount of time, which is obviously problematic in situations without any demand for fluid, for example, causing the tank to be filled faster than usual. The controller would therefore always have to assume no demand in order to be on the safe side, a limitation that is unnecessary when feedback is available. To keep the case study simple, however, fluid removal is assumed to be constant, making the feedback provided by the sensor unnecessary for reasons of control. But as soon as faults such as $\neg\mathsf{timeout}$ are considered, the sensor's feedback is indeed relevant as it adds a level of redundancy to the system. The feedback therefore helps to increase system safety by reducing the chance of incorrect control decisions that would result in tank ruptures.

3.1.3 Zero Execution Time

The behavior of discrete-time systems is given by a sequence of system steps, where each step corresponds to a tick of a logical clock with a fixed tick rate Δt. The system's real-time behavior is therefore abstractly represented by discrete-time behavior based on the logical clock. Often, the assumption of zero execution time is made when developing reactive systems [20, 119] that further allows to abstract from the system's actual timing behavior to simplify modeling and analysis: The systems are assumed to react infinitely fast to any input stimuli, instantly computing their new states and outputs. Figure 3.4 visualizes this concept that it is particularly useful to model controllers: In zero time, the controllers observe changes in the plants via feedback, compute the appropriate control actions, and forward the computed actions to the plants.

The zero execution time assumption is justified and adequate only if the actual real-world system indeed turns out to be faster than its environment, always producing and emitting its outputs before its environment is able to generate new inputs. State changes and output generation are only then guaranteed to be completed before any new inputs are available, thus making it possible to abstract from the system's precise reaction times. For example, an electrical circuit stabilizes much faster than a human can switch a button. In the height control case study, the decision for a tunnel closure is made in

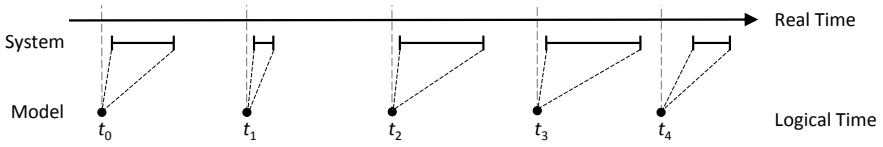


Figure 3.4. Illustration of the zero execution time assumption: The real-world system takes some amount of time to react to each input stimuli occurring at different, non-overlapping points in time. Depending on the inputs and the system's current state, the length of these response intervals can vary. The model, on the other hand, abstracts the response intervals into distinct zero-time response instants over logical time.

a tiny fraction of the time it takes a vehicle to drive from the last sensor to the tunnel entrance. Hence, it can reasonably be assumed that the height control is able to close the tunnel before the vehicle collides, even though the actual sensor readouts, decision computations, and the tunnel closing procedure do in fact take some time. For many systems it is therefore valid to assume reactions to be instantaneous, simplifying system modeling and making formal analyses more efficient. To ensure adequacy, techniques and tools [205] are available to validate that the worst case execution times of all real-world system reactions do not exceed the step interval Δt used during analysis. For software-based systems, it is also possible to rely on implementation languages whose semantics are built upon the assumption of zero execution time; Esterel and Lustre are two examples for such languages that are used in practice [20].

3.1.4 Safety-Critical Systems

Safety-critical systems consist of one or more controllers that monitor one or more plants, intervening if necessary to prevent potentially bad plant behavior that might result in hazards; hence, safety is a control problem [131]. Both plants and controllers are systems in the systems-theoretical sense, therefore the distinction between both is a question of perspective: A controller can itself be seen as a plant that can be controlled by another controller as illustrated by Figure 3.5. A plant is controlled by an automated controller through sensors and actuators that observe and influence the plant, respectively. Both the plant and the automated controller can be additionally controlled by a human that operates the automated controller via a keyboard, for instance, and observes its outputs on a graphical display. From the human's point of view, the automated controller is thus also a plant. Humans are typically better suited to react to completely unforeseen circumstances than automated controllers [131], and therefore often have the possibility to override the actions carried out by the automated controllers, if necessary. On the other hand, they have to keep two models in their head, one for the inner workings of the plant and one for the automated controller. Operational faults by a human controller can therefore also cause hazards; standard operational procedures, good training, and experience, for instance, help to avoid such human mistakes [131]. Figure 3.6 illustrates the classification of components into sensors, controllers, actuators, and plants as well as the feedback between them for models of safety-critical systems. Depending on the system's complexity and the model's level of abstraction, however, additional layers of controllers might be necessary to model systems like the one represented by Figure 3.5.

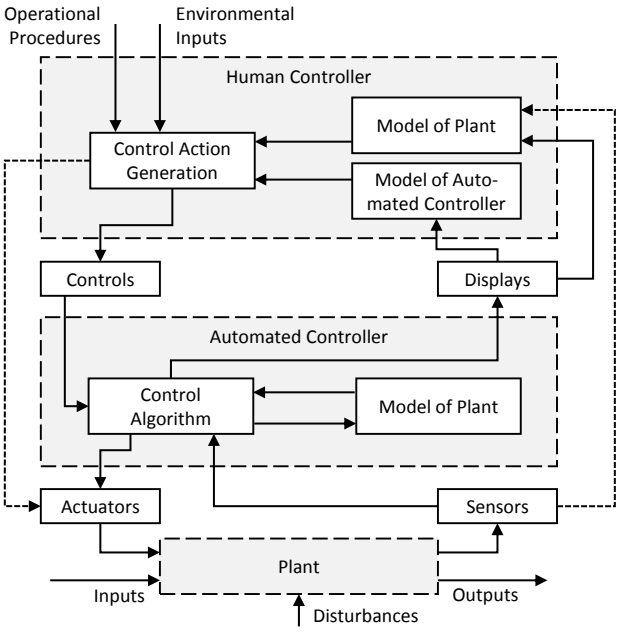
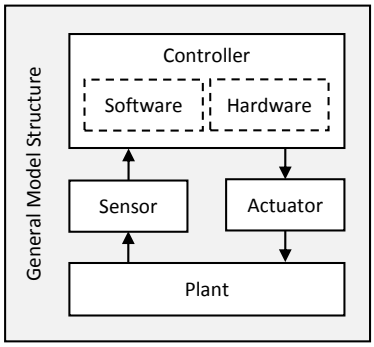


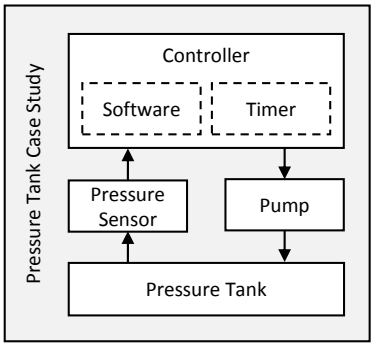
Figure 3.5. Overview of the general structure of safety-critical systems [131]: The plant is controlled by an automated controller, both of which are in turn controlled by a human. The plant can be affected by external disturbances that typically have negative impacts on the plant's behavior. The automated controller indirectly observes the state and behavior of the plant using available sensors, utilizing the sensed data as inputs for its control algorithm and to update its internal model of the plant. The control algorithm predicts the plant's future behavior with the help of the internal model, if necessary using the actuators to influence the plant to prevent possible future undesired behavior. Displays and input devices such as keyboards are the actuators and sensors that allow a human to observe and control the automated controller.

Plants. Plants can either be part of the safety-critical system under development or they merely represent some parts of the system's environment, i.e., the context that the system is embedded into. In the latter case, the plant is not built and developed, instead it is simply assumed to exist; the system's task, however, is still to control the plant to prevent hazards for as long as possible. For example, the pressure tank is part of the system under development, whereas the overheight vehicles in the height control case study are not. In fact, the height control does not influence the vehicles, but the humans driving those vehicles instead. Humans are notoriously hard to predict and model adequately, often requiring nondeterministic abstractions. In general, however, neither the modeling approach nor the model-based safety analysis techniques concern themselves with the kind of plants that are controlled, i.e., whether the plants consist of humans, preexisting technical systems, newly developed technical systems, or some mixtures thereof. The plants must only be modeled in sufficient detail to specify the hazards and to adequately describe the feedback between plants and controllers.

Sensors & Actuators. The plants represent the entirety of what the controllers are able to observe and influence. Controllers use some form of sensors for the former and some kinds of actuators for the latter. For instance, a pressure sensor and a pump are



(a) The general structure of models of safety-critical systems. The controller is often a mixture of both hardware- and software-based components; it too can be controlled by other controllers, either automated or human, as illustrated by Figure 3.5.



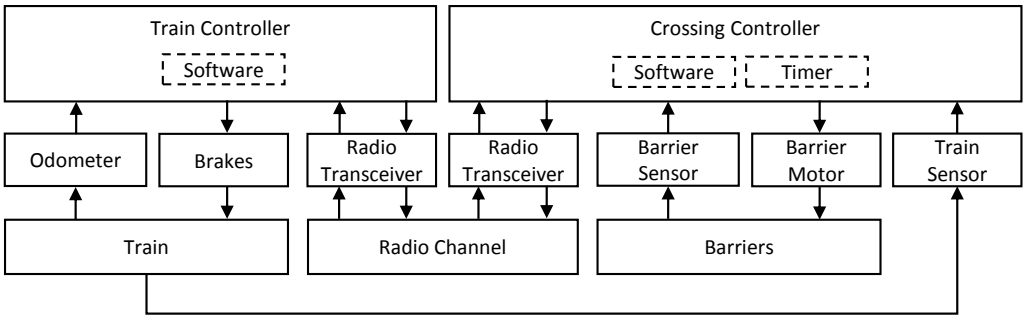
(b) The model of the pressure tank case study follows the general model structure. The pressure tank is the plant that is controlled; its pressure level is observed by the pressure sensor and influenced by the pump that is enabled and disabled by the controller.

Figure 3.6. Overview of the basic structure of models of safety-critical systems [131, 186] in general (left) and specifically for the pressure tank case study (right).

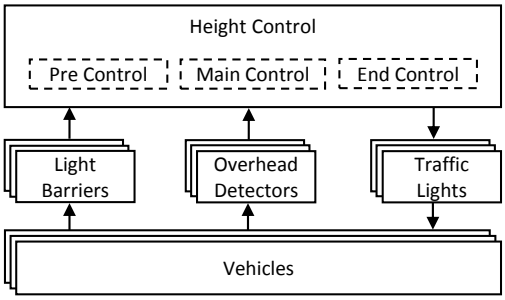
used by the controller of the pressure tank case study to observe the pressure level within the tank and to start or stop increasing it, respectively. Human controllers, on the other hand, typically use switches, buttons, and control lamps to control analogous plants. For digital plants, they use digital input methods such as keyboards and mice as well as screens and displays for observations. However, all of these different types of sensors and actuators only allow indirect observations of and indirect influence on the plants. In particular, sensor faults either prevent the controllers from observing the plants altogether or they provide incorrect observations, whereas actuator faults either take away the controllers' ability to influence a plant or they change the actuators' effects on the plants.

Controllers. Controllers make use of feedback to cope with faults originating from either the controllers themselves, the plants, the sensors, or the actuators. Additionally, feedback enables the controllers to determine whether their control actions are effective, that is, whether the plants are influenced in a desirable and intended way. To do so, feedback is used to update the controllers' internal models of the controlled plants, which in turn facilitates predictions of possible future plant behavior. Controllers represent all of the means available to the system under development to influence the plants, often making use of various different kinds of components: Just like plants, controllers typically consist of mixtures of digital hardware and software components as well as analog parts such as electronic circuits or hydraulic systems.

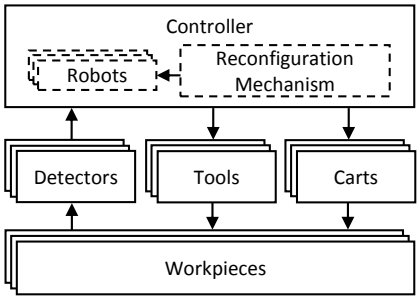
Application to the Case Studies. For some of the case studies, the basic classification of system components into plants, controllers, sensors, and actuators is shown in Figures 3.6b and 3.7. Some of the models contain multiple controllers and/or multiple plants that are observed and influenced by different kinds of sensors and actuators. The robot cell case study even has nested controllers in the sense that the system's reconfiguration mechanism can be seen as a controller for the robots and carts.



(a) In the radio-controlled railroad crossing case study, there are three plants: The train that approaches and passes the crossing, the barriers that secure the crossing, and a radio channel that is used by the radio-based communication and synchronization of the two physically separate train and crossing controllers. All three plants are preexisting technical systems that are therefore not developed in the context of the case study, but are slightly modified to allow control by the train and crossing controllers. The train controller observes the train's position and speed using an odometer and brings it to a halt during emergency stops by activating the train's brakes. The crossing controller uses a sensor to determine whether the barriers are open or closed and activates the barrier motor to close or open the barriers. Both controllers make use of radio transceivers to communicate with each other over the radio channel; the radio transceivers are considered to be both sensors and actuators simultaneously.

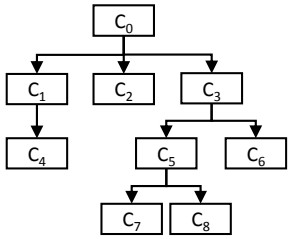


(b) In the height control case study, the controlled plants are the vehicles driving through the Elbe Tunnel. While the light barriers and overhead detectors observe the vehicles' positions and, in particular, the lanes they drive on, the traffic lights actually influence the human drivers within the vehicles instead of the vehicles themselves. The overall height control consists of three subcontrollers responsible for observing different sections of the road leading to the tunnel entrance. In contrast to the two controllers of the railroad crossing case study, however, they are not physically separated but instead represent a decomposition of the software constituting the height control to increase modularity.

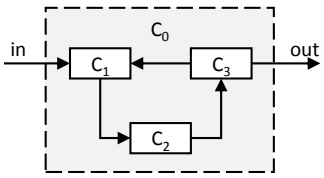


(c) In the self-organizing robot cell case study, the workpieces are the plants. The carts transport the workpieces between different robots for processing as well as into and out of the overall production cell. Robots use their tools to modify the workpieces in order to complete the tasks that the workpieces require. Workpiece positions are tracked through a set of detectors, allowing both robots and carts to determine whether they can interact with the workpieces. The reconfiguration mechanism is a nested controller that observes the robots and carts, changing their configurations if they can no longer fulfill a workpiece's task due to a fault, for instance.

Figure 3.7. Overview of the model structures induced by control theory for the railroad crossing, height control, and robot cell case studies. In each subfigure, plants are shown at the bottom, whereas controllers are at the top. Sensors have arrows leaving the plants and going into the controllers to indicate observation of plants, whereas the direction of the arrows is reversed for actuators to represent interferences with the plants.



(a) An exemplary hierarchical decomposition of the component C_0. Some of its subcomponents such as C_3 are further decomposed whereas C_2 does not require further decomposition for some reason.



(b) An exemplary view of component C_0 from the left, showing its direct subcomponents and the interdependencies between them that allow C_0 to compute its outputs for its inputs.

Figure 3.8. An illustration of hierarchical system decomposition (left) that allows the root component C_0 to be modeled independently of the rest of the system (right).

3.2 Modeling System Structure

Models of system structure describe the set of components a system consists of as well as their interrelationships [25, 186]. To cope with complexity, system components are decomposed into more elementary, less complex subcomponents, iteratively resulting in a decomposition of the component hierarchy as illustrated by Figure 3.8a [203]. The leaves of the hierarchy, that is, C_2, C_4, C_6, C_7, and C_8, represent components for which further decomposition is not required for one of two reasons: Either the components are modeled in sufficient detail to be ready for implementation in hardware or software. Alternatively, they might be standard off-the-shelf components that can be bought from a third-party vendor and incorporated as-is into the final system such as the pump or the pressure sensor of the pressure tank case study. In the latter case, the component vendors already determined all possible component failures using some safety analysis technique, most likely employing a similar decomposition approach. During safety analyses of the larger system under development, these failures are used in conjunction with an abstract model of the third-party component's behavior [186].

Most models of safety-critical systems make use of such abstractions for off-the-shelf components. Additional abstractions regarding the actual system's structure or behavior are often required to increase the model's comprehensibility and to facilitate formal analysis [13, 43, 90, 106]: Details not deemed to be relevant for the purpose of the model are omitted. Hence, abstractions are crucial to reduce model size and complexity, but they might also make a model useless for its purpose. In general, which abstractions are made is somewhat arbitrary and whether the abstractions fulfill their intended purpose without compromising adequacy can only be checked through verification and validation. The exclusive use of discrete-state, discrete-time models throughout this thesis is an abstraction, for instance, which potentially threatens adequacy due to the approximation errors of the numerical integration methods.

Ideally, the individual components of a safety-critical system are modeled as separately and independently as possible both to decrease model complexity as well as to increase model reusability. Components that are not inherently and explicitly tied to other concrete components are often easier to understand, model, develop, and maintain

as long as their assumptions about and their guarantees for other components are known and specified. In this sense, modeling safety-critical systems is no different than software modeling and systems development in general, that is, fundamentals such as modularization, encapsulation, low coupling, high cohesion, the open/closed principle, etc. all apply just as well [161]. However, while the different components can be modeled and developed separately, they can only be fully analyzed together: Not only the individual component behavior potentially affects system correctness and system safety, component interactions within a system can do so as well. Undesired, faulty interactions of components that individually behave as intended indicate a systematic design flaw or a communication fault that can potentially result in hazards; with increasing system complexity, such interaction faults become more likely [131].

SysML block definition diagrams as well as internal block diagrams allow for structural modeling and system decomposition at various levels of abstraction [33, 96, 153]. The former describe the individual blocks a model consists of. A block denotes all types of things within a system or its environment that are of relevance, such as real-world entities like humans or other systems, larger hardware or software subsystems of the system under development, individual software classes, microprocessors, hydraulic or electrical components, and so on. Therefore, the notion of component used so far captures just a small part of what blocks are actually able to represent; nevertheless, the following often uses the two terms interchangeably for reasons of simplicity. Block definition diagrams by themselves are insufficient to completely describe all structural aspects of a system. Instead, they must be used in conjunction with internal block diagrams that, as implied by their name, show the internal structure of a block, similar to how Figure 3.8b illustrates the inner workings of the root component of Figure 3.8a. Consequently, block definition diagrams show the individual parts a system can be composed of, whereas internal block diagrams describe how the blocks are used, instantiated, and connected to compose other blocks and, eventually, the overall system.

3.2.1 Structural Modeling Guidelines

For formal safety analyses, it is important that the models reflect the general control-theoretical system partitioning into plants, controllers, actuators, and sensors in order to completely capture the entire feedback cycle between these different parts of the system. The first core modeling guideline for safety-critical systems is therefore inferred from control theory:

Guideline 1 (Separation of Plants and Controllers)**.** *Models of safety-critical systems should clearly separate plants and controllers, also denoting the sensors and actuators available to the controllers for observing and influencing the plants.*

A controller internally has an implicit or explicit model of the plant that it controls in order to reason about the plant's state and to predict the plant's future expected behavior. It influences the plant using the actuators it has available to indirectly change the plant's state and therefore its future behavior. The controller uses all sensors it has access to in order to observe and monitor the plant, trying to deduce the plant's current state. As the controller can only indirectly observe the plant using its sensors,

discrepancies can emerge between the controller's perceived state and the plant's actual state. Typical reasons for such discrepancies are faults or unexpected environmental disturbances that cause the controller to receive incorrect sensor data or that result in unintended actuator effects on the plant. State discrepancies might subsequently result in incorrect controller actions, as the controller is likely to predict the plant's future behavior incorrectly. Therefore, it might accidentally and unknowingly execute destructive control actions, or it might not take an action that it should have taken. Such control failures do not necessarily result in hazards, however: Due to the feedback the controller receives from the plant via the sensors, it might be able to discover and resolve the state discrepancy, aligning its perceived state with the plant's actual state before it is too late to prevent a safety-critical situation. Consequently, these state discrepancies must be part of the analyzed model in order to obtain adequate safety analysis results.

In the pressure tank case study, for instance, the pressure sensor's $\neg$is full fault makes the controller believe that pumping can safely continue whereas in reality, the tank is about to rupture. If the timer's $\neg$timeout fault also occurs, the tank ruptures, as the controller has no way of observing the tank to be full and consequently does not shut off the pump. If, on the other hand, the timer does in fact signal a timeout, the controller is able to update its internal knowledge about the pressure level before it is too late and it can then disable the pump just in time to prevent a tank rupture.

Guideline 2 (Specification of Hazards)**.** *It should be possible to exclusively specify hazards over the models of the controlled plants, thereby ensuring that all influences on system safety are considered during formal safety analysis.*

As hazards typically arise from an inability to control negative influences on the system, hazards represent control failures that safety analyses are intended to identify [131]. The specification of such failures generally cannot be expressed in terms of the control mechanisms, but solely in terms of the controlled plants as the controller usually is not even aware that it failed. In general, violations of properties of the form "for some plant behavior b, the controller should perform some action a" are issues of functional correctness which obviously can be safety-critical in many cases. As for components, however, only considering individual behavior and functional correctness is insufficient: Hazards often arise due to unintended and more concealed feedback loops between a plant and its controller, in particular when the controller misinterprets the plant's behavior or its control actions do not have the intended effects on the plant due to actuator faults, for instance. Specifying hazards in terms of the plants allows formal safety analysis techniques to consider both sources of control failures uniformly. In the pressure tank case study, for example, the pressure tank is the plant with the hazards of tank ruptures and complete tank depletions specified exclusively over the tank's state. For the formal safety analysis techniques, it is irrelevant whether a tank rupture is the result of an incorrect controller action because of a design mistake or the result of an omitted controller action due to $\neg$is full and $\neg$timeout. Consequently, Guideline 2 underlines the importance of Guideline 1, requiring that all necessary information for the adequate specifications of the hazards is contained in the plant models.

Guideline 3 (Structural Modularity & Composability)**.** *A modular modeling approach should be followed that supports system decomposition, block encapsulation, and block*

composition to reduce modeling complexity and to conveniently model and analyze several combinations of system design alternatives.

A modular modeling approach helps to identify and fix modeling or specification flaws as it makes it easier to consider blocks in isolation. Modular models increase comprehensibility and composability, resulting in fewer development faults that otherwise might manifest themselves as hazards during system operation [25, 131]. Guideline 3 therefore motivates the use of SysML's component-oriented modeling techniques to iteratively decompose safety-critical systems. Modeling best practices [161] such as encapsulation, separation of concerns, or the single responsibility principle, for example, help to ensure modularity and therefore should be adhered to whenever possible. Modularity and composability are closely related, as the same modeling principles that support modularity in general also increase composability. Clear interfaces between different components and, in particular, adherence to the Liskov substitution principle [161] further support composability and consequently help to follow Guideline 3. There are several reasons why flexibility in model composition is beneficial: Especially during early phases of development, different design variants of a system are often evaluated before one is chosen and actually developed. Moreover, safety analyses might show the necessity for additional safety measures in order to decrease hazard probabilities, but their implementations must in turn also be checked for additional safety issues. Finally, safety-critical product lines are also gaining more traction [16] where multiple different variants of a safety-critical systems are assembled based on customer wishes.

3.2.2 Component Modeling

SysML blocks are a very general modeling concept with very few constraints restricting their use for modeling. Similarly, the systematic modeling approach for safety-critical systems is very liberal regarding both structural and behavioral modeling. In particular, a strict hierarchical decomposition is not enforced as it is sometimes beneficial for some blocks of a model to be able to reference other blocks without any physical containment relationship existing between them. For instance, the modeling approach allows for the train controller of the railroad crossing case study to reference the physically separate crossing controller directly. For highly abstract models, this structure might be adequate to abstract from the radio-based communication between the two controllers. In less abstract models, by contrast, such direct references must be avoided for reasons of adequacy, instead requiring a radio channel that handles the communication between the two controllers as shown in Figure 3.7a. The systematic modeling approach therefore only gives some basic guidelines and restrictions on how SysML's modeling flexibility is best used for model structuring. In the end, adequate structuring depends on both the system to be modeled as well as the purposes the model is created for.

SysML blocks are defined as a stereotype for UML classes [153]. Blocks therefore support all features that UML provides for classes, such as generalization, interfaces, properties, associations, and operations; value types and enumerations can also be declared in SysML block definition diagrams [33, 153, 154]. Often, blocks have internal state that, at least ideally, they hide from other blocks to follow the principle of encapsulation [161]. Blocks can be composed of other, more elementary ones to decompose models along their

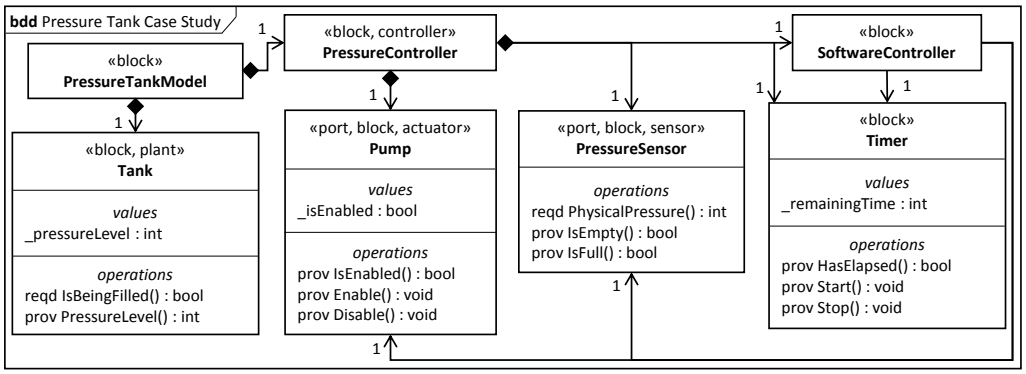


Figure 3.9. The SysML block definition diagram for the pressure tank case study showing all of the blocks the model consists of as well as their state values, ports, required and provided operations, and associations. As per Guideline 1, the overall model is separated into a controller and a plant as indicated by the custom stereotypes «controller» and «plant», respectively. The Pump and PressureSensor blocks are also clearly labeled with the «actuator» and «sensor» stereotypes; they are ports of the overall PressureController, allowing the controller to observe and influence the plant. Guideline 3 is satisfied due to the modular modeling of the individual components constituting the case study. The analyzed hazards can be completely specified over Tank::_pressureLevel alone, adhering to Guideline 2.

physical or logical structure. They are allowed to reference arbitrary other blocks they depend on through SysML's notation for associations and dependencies. Additionally, blocks can declare provided and required operations that handle the communication between them; two such operations of the same signature can be connected in an internal block diagram, with the connection triggering the behavior associated with the provided operation whenever the required operation is used. A block definition diagram containing all of the blocks of the pressure tank case study is shown in Figure 3.9.

Block Relationships. Blocks can be associated with other blocks in multiple ways: Standard UML associations indicate that a block is somehow able to reference and access other blocks. Composite associations, by contrast, model part-whole relationships where the part is physically or logically owned by the parent block but still has an identity of its own [153, 154]. In the pressure tank case study, for example, the Pump, PressureSensor, Timer, and SoftwareController are all parts of the overall PressureController as illustrated by Figure 3.9, whereas the SoftwareController is simply associated with the other three aforementioned blocks. Using these references, the software reads sensor values, checks for timeouts, and enables or disables the pump. For formal safety analyses of the case study, however, the exact sequences of real-world operations that somehow enable the software to read out and control the hardware components is deemed to be of little importance. Thus, these associations abstract away the drivers, APIs, operation system, and digital to analog/analog to digital converters that give the actual controller software access to the respective real-world hardware components.

State. A block's values and association ends correspond to a part of the system's state vector, whereas the overall state vector of the system is given by the entirety of all block values and association ends. The operations and state machines associated with the

blocks, cf. Section 3.3, make up the state equations that change the system's state and therefore describe its behavior. In Figure 3.9, for instance, the value Tank::_pressureLevel denotes the state of the pressure tank block, i.e., the pressure level within the tank. Similarly, Timer::_remainingTime and Pump::_isEnabled indicate how much time is left until a timeout occurs and whether the pump is enabled or disabled, respectively. PressureSensor, on the other hand, has no state. The state of the SoftwareController is hidden behind its associated state machine as discussed in Section 3.3.

Ports. The «port» stereotype of the Pump and PressureSensor blocks in Figure 3.9 indicate that these two blocks are ports of the PressureController block. Ports allow for indirect communication between blocks without the blocks knowing about each other. In SysML, ports are always typed by blocks that declare provided and required operations for other blocks to access. Provided operations can be seen as declarations of services that a block provides for consumption by other blocks. Required operations, on the other hand, are services that a block requires from other blocks to perform its intended function. Operations that a block requires must therefore be satisfied by operations that another block provides when assembling a system. Figure 3.9 shows the operations that the individual blocks provide or require, like, for instance, PressureSensor::IsFull or PressureSensor::PhysicalPressure, respectively. The former is used by the control software to check whether the tank is full and the pump must be shut off. The latter, by contrast, is a required operation that allows the pressure sensor to determine the actual pressure level within the tank. Such connections between ports are not shown in block definition diagrams; they are established by internal block diagrams when larger blocks or the overall system are assembled out of more elementary blocks.

3.2.3 Component Composition

Block definition diagrams such as the one in Figure 3.9 only show the blocks and their relationships that make up the overall system model. The actual block and system composition, on the other hand, can only be shown by internal block diagrams like the one in Figure 3.10 for the pressure tank case study. Just like objects are instances of software classes, blocks represent types of things, that is, structural specifications of sets of block instances. Thus, internal block diagrams show the block instances and the connections between their ports that constitute other blocks or the overall system.

Port Connections. Required and provided operations of the same signature, that is, operations with matching return types as well as parameter types, can be connected, resulting in invocations of the required operation to be forwarded to the connected provided operation. In the pressure tank case study, for example, the tank::IsBeingFilled required operation is connected to the pump::IsEnabled provided operation, meaning that whenever the tank needs to determine whether it is being filled, the pump checks whether it is enabled. As the tank is only indirectly connected to the pump, there can be other system configurations with completely different connections to determine whether the tank is being filled. If this level of indirection is not desired, direct connections between an operation and block can be established as shown for the software and the timer. In these cases, the corresponding conjugated operations are either not modeled for reasons of brevity, or the connections represent invocations through reference associations.

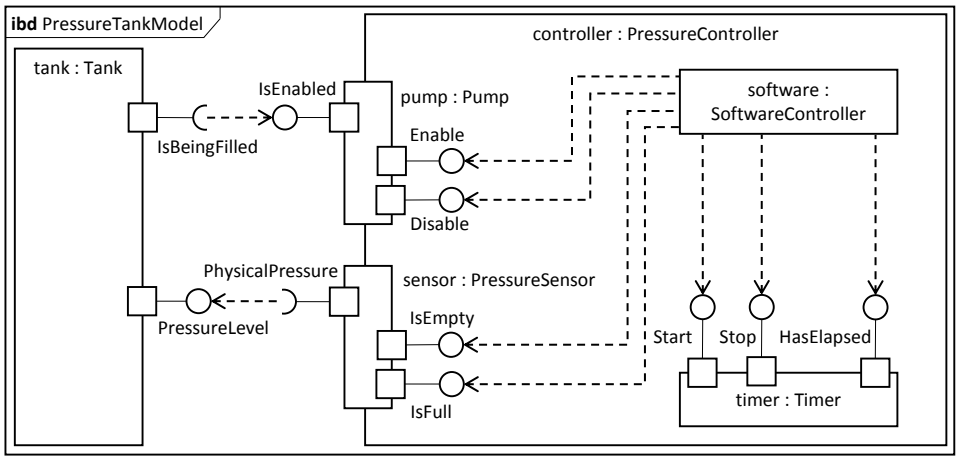


Figure 3.10. The SysML internal block diagram for the pressure tank case study model showing how the various blocks of Figure 3.9 are composed together. The pump and sensor instances are shown as nested ports on the controller, with their IsEnabled and PhysicalPressure operations connected to the tank's IsBeingFilled and PressureLevel operations, respectively. Required and provided operations are shown with SysML's balls and sockets notation, which actually is an abuse of notation; SysML generally only allows block-typed ports, which, however, would introduce too much syntactical overhead for this simple model with no benefit. The provided operations directed inwards into the controller are directly used by the SoftwareController instance through its associations with pump, sensor, and timer.

Modeling Design Variants. With SysML's support for modular modeling and flexible block composition, additional abstractions and levels of indirection can be added to a model to allow for the composition of different system design variants as shown in Chapter 7 for the height control case study. Another example is a more reliable pressure detection mechanism based on the triple modular redundancy safety pattern [6, 25] that increases the pressure tank case study's fault tolerance: With triple modular redundancy, the critical component is replicated three times and a voting mechanism is introduced that determines the actual outputs through majority voting or other policies. As a result, fault tolerance improves because a failure of only one of the components no longer results in incorrect outputs. Figures 3.11 and 3.12 show the application of the triple modular redundancy pattern to the pressure tank case study's pressure sensor: The PressureSensor block from Figure 3.9 is replaced by a new pressure detection block TripleModularRedundantDetection that incorporates three instances of either the original pressure sensor or nested triple modular redundant detectors, plus a voting mechanism to determine the resulting values that should be returned by the IsFull and IsEmpty provided operations. Consequently, safety improves as now at least two instances of ¬is full must occur in addition to ¬timeout in order to cause a tank rupture.

As shown by Figures 3.11 and 3.12, the two software design patterns Composite and Strategy can be used to model the improved version of the case study [66]. Following the Composite pattern, a common interface IPressureDetector is introduced that abstracts away the differences between the original design and the triple modular redundant one. For the voting module, an IVotingStrategy interface is introduced that provides a

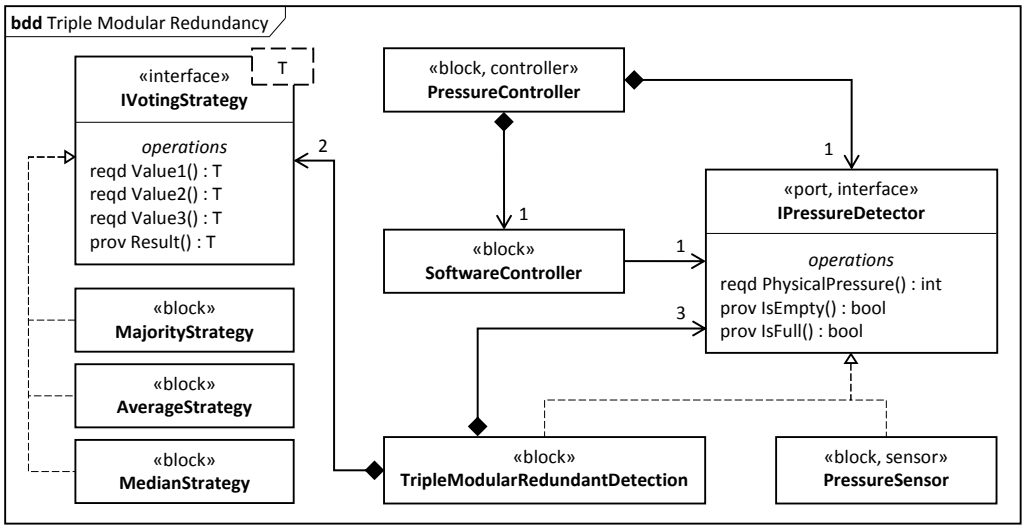


Figure 3.11. A partial block definition diagram showing how the reliability of pressure level detection can be improved through the use of triple modular redundancy. The PressureSensor block and the TripleModularRedundantDetection block realize the same IPressureDetector interface in order to allow them to seamlessly replace each other, improving model modularity and composability. Multiple voting strategies are abstracted away behind a common IVotingStrategy interface; a non-exhaustive set of three strategies is shown. The unchanged parts of the PressureController like the Pump or the Timer are omitted.

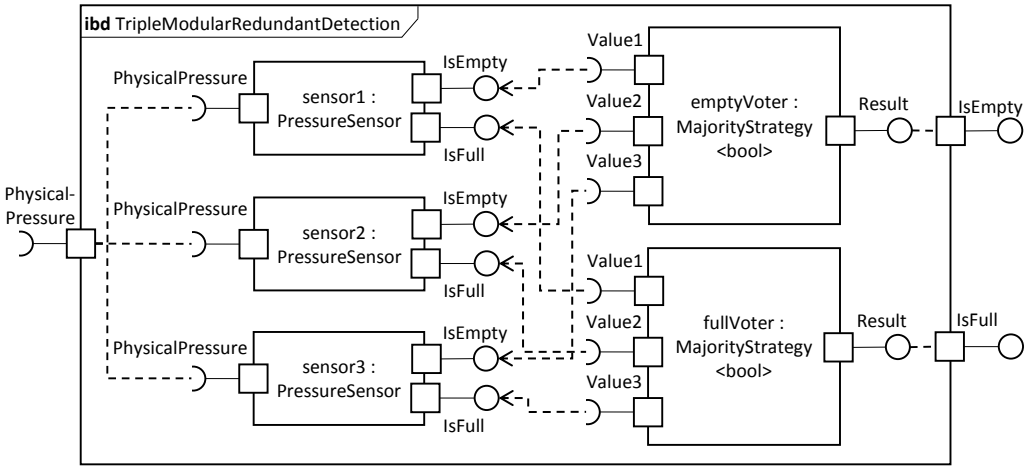


Figure 3.12. An internal block diagram showing an instance of the TripleModularRedundantDetection configured to use MajorityStrategy instances to determine the value of the IsEmpty and IsFull ports. Delegate connections are used to forward the voter's Result ports to the overall detector's IsFull and IsEmpty ports in addition to distributing the incoming PhysicalPressure to the three PressureSensor instances the overall detector consists of. Connections are established between the sensor*n*::IsEmpty provided operations and the emptyVoter::Value*n* required operations as well as between the sensor*n*::IsFull provided operations and the fullVoter::Value*n* required operations for $n \in \{1, 2, 3\}$. As the sensors within the detector are typed as IPressureDetector, multiple TripleModularRedundantDetection could be nested for further reliability improvements.

consistent surface area for different voting mechanisms such as majority voting or average computations. The voting mechanisms are independent from the PressureSensor blocks through the use of ports and port connections, making it possible to reuse them in other situations as well. Overall, this setup allows for modeling and analyzing various different system variants, including the original one shown in Figure 3.10.

3.3 Modeling System Behavior

There are several behavioral modeling paradigms that can be used to describe the behavior of a system: Data flow, control flow, state machines, or state equations, for example [20, 145, 153]. Data flow focuses on functional dependencies between system inputs and outputs, linking the outputs of a step to the inputs of the next steps. Control flow, by contrast, clearly shows the sequences of operations as well as the conditions under which certain operations are executed, deemphasizing the connections between inputs and outputs. State machines model discrete sequences of system state changes over time; as in systems theory, a state encompasses all of the information that is necessary for the specification of the block's current and future behavior, with transitions changing the block's state. State equations are most useful to describe the correlation between different physical quantities.

Each of these behavioral modeling paradigms has its strengths and weaknesses in modeling certain behavioral aspects. For instance, while it is possible to model sequential operations using state machines, the resulting model is less concise and therefore harder to understand than a control flow-based one. It is therefore often beneficial to mix several paradigms, modeling the behavior of different components or different behavioral aspects of the same component using multiple paradigms. The following mostly focuses on behavioral modeling with state machines and control flow as the case studies can be adequately modeled without data flow or differential equations. The physical quantities that are indeed important for the case studies are approximated using numerical procedures as previously discussed; SysML parametric diagrams describe the correlations between the different physical quantities in these cases.

Such approximations of differential equations are a form of behavioral abstraction [25, 127]. Other forms of behavioral abstractions also play a crucial role in behavioral modeling: The leaf components in Figure 3.8a on page 31 are abstractions of the actual components they represent, because even though they could have been decomposed even further, the chosen system decomposition was deemed to be appropriate for the purposes of the model. The behavioral models of the leaf components consequently have to abstract away how the omitted subcomponents behave and interact with each other. Even for intermediate components within the hierarchy that are fully decomposed, behavioral abstraction is sometimes necessary or helpful. For instance, the exact behavior might be unknown or too complex to specify especially when humans are involved, or it might unacceptably slow down analyses. Multiple imaginable behaviors are another potential reasons when the one that will eventually be built into the actual system is not yet decided during early phases of development. In all of these cases, the actual behavior is typically abstracted away using nondeterminism.

3.3.1 Behavioral Modeling Guidelines

There are interdependencies between behavioral modeling and the structural decomposition into blocks. If the system structure is modeled inappropriately, behavioral modeling becomes cumbersome and reduces model comprehensibility. Conversely, good model structuring usually supports behavioral modeling. Models are therefore created iteratively to cope with the interdependencies between structural and behavioral aspects of the system: The first iteration structurally decomposes the system before actual component behavior is modeled, which in turn potentially requires structural changes and thus additional iterations. The primary goal of structural modeling is thus to support and ease behavioral modeling as formal safety analyses are mostly concerned with reasoning about behavior, whereas structure only plays a secondary role. Following the structural modeling guidelines is still important, however, as model adequacy or comprehensibility might be compromised otherwise.

Guideline 4 (Behavioral Modularity & Composability)**.** *Behavior should be modularized and modeled for each component individually to allow for behavioral composability.*

One of the main reasons for system decomposition is the modularization of behavior, allowing individual components to be modeled separately. The basic principle of encapsulation not only refers to state, but also to behavior [161]: Components expose only their "what", but not their "how" [203], meaning that contracts are defined over the components' ports that describe what the components do, but abstract away how the components actually achieve their intended behavior. Consequently, established modeling principles such as design to interfaces, polymorphism, and the Liskov substitution principle also apply when modeling safety-critical systems [161].

Guideline 5 (Adequate Modeling Paradigms)**.** *Behavior should be modeled using one or more behavioral modeling paradigms that increase model comprehensibility and adequacy while still allowing for formal analyses.*

Ideally, the most appropriate modeling paradigm would be used for each behavioral aspect of a system, combining multiple ones to achieve the highest possible model quality. However, mixing too many different paradigms significantly complicates formal analyses due to the increased complexity of the modeling language semantics, potentially making fully exhaustive model checking infeasible [58, 118]. It is therefore necessary to find a tradeoff that balances formal analysis applicability and efficiency on the one hand and the model's level of expressiveness and comprehensibility on the other hand. The modeling approach outlined in this chapter, for example, abstracts from continuous quantities that are often best specified using differential equations. For systems where only a small fraction of the component behavior is continuous, discretizations using numerical approximations often make the modeling approach applicable nevertheless; for predominantly continuous systems, however, the approach is likely inappropriate.

Guideline 6 (Feedback)**.** *Zero-time feedback loops between plants and controllers should be avoided as they likely violate the zero execution time assumption.*

Despite the assumption of zero execution time, feedback loops are usually required to take time. That is, the controllers observe the plants, compute the appropriate control actions, and use the actuators to influence the plants in zero time, while the plants'

reactions to the control actions can only be observed at a later point in time due to inertia. It is therefore necessary to identify the feedback loops that exist within a system model and, if necessary, to break up any zero-time loops by adding time delays. For systems that are exclusively modeled using differential equations, it must be possible to find a solution; for control flow- or state machine-based behavior, a fix point must eventually be reached in which the overall system state is stable until the next inputs arrive. In general, zero-time feedback loops can usually be broken up by linearizing the components the loop consists of and adding a non-zero-time delay appropriately. Especially at the level of abstraction required to formally analyze safety-critical systems, it is often sufficient to delay the controller actions by one system step before they reach the plants again in order to break up feedback loops. Some implementation languages like Esterel have complex semantics that allow them to automatically determine whether zero-time feedback loops exist [189]; they still must be fixed manually, however. Similarly, real-time critical robotics applications require the use of special primitives to break up feedback loops that take no time [196].

Guideline 7 (Closing the Models)**.** *Behavior should not depend on any inputs that are not explicitly modeled to increase analysis efficiency.*

The systems-theoretical characterization of safety-critical systems allows for arbitrary system inputs and outputs, some of which are part of the feedback loop when a controller is introduced to monitor and control a plant. Inputs that are not determined by some outputs, however, can be problematic for formal safety analysis techniques: As the model contains no information for these inputs, they must be assumed to be completely nondeterministic, increasing analysis complexity and therefore reducing analysis efficiency. Some completely nondeterministic inputs are unavoidable, for instance to model faults and to abstract human behavior when it cannot be predicted. Other kinds of nondeterministic inputs hint at underspecification, however, making safety analyses slower and inaccurate or possibly even inadequate.

3.3.2 Model of Computation

Established component- and object-oriented modeling techniques and best practices are usually sufficient to create behavioral models complying with Guideline 4. Guidelines 5 to 7, by contrast, are primarily concerned with the creation of adequate and comprehensible models that can still be formally analyzed, two goals that are sometimes at odds with each other. The model of computation that underlies all system models throughout this thesis tries to strike a balance between these two somewhat antagonistic goals. It depends on discrete formalizations of system states, without any explicit support for real-time modeling or continuous variables; such continuous quantities have to be discretized manually using numerical procedures as discussed before. Consequently, the systematic modeling approach outlined in this chapter has to be adapted for systems where such a discretization would be inappropriate, also requiring other formal analysis tools that support timed or hybrid system models. However, for system models where neither timed nor continuous behavior plays an important role and can be adequately abstracted from, the discrete-state, discrete-time model of computation allows for efficient safety analyses as discussed in Chapters 5 and 6.

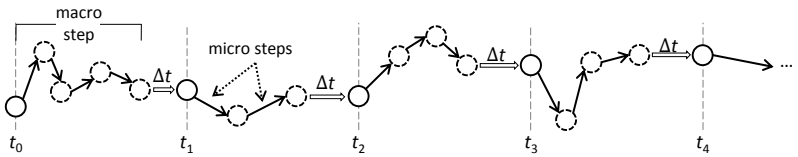


Figure 3.13. Illustration of the micro-macro steps semantics underlying the model of computation: Solid states drawn at the logical time boundaries $t_0, t_1, \ldots$ are actually observable, while the intermediate dashed states and micro steps are unobservable, system-internal computations that are invisible from outside the system. The macro step beginning at time t_0, for instance, consists of four micro steps, resulting in the macro step's final state in zero time. From this state, a new macro step starts at time t_1 after the step duration Δt has passed.

Micro & Macro Steps. As illustrated by Figure 3.13, the model of computation implements the zero execution time assumption for reactive systems by distinguishing between micro and macro steps, a distinction that is also made by the synchronous programming language Esterel as well as by Harel's transition system formalism [31, 169, 189], for example. Macro steps describe the observable behavior of a system, whereas the micro steps illustrate how and why the behavior occurs, e.g., individual steps of control flow behavior or individual state machine transitions. Overall, the system is seen as executing a sequence of macro steps beginning at fixed points in time $t_0, t_1, t_2, \ldots$, with each macro step assumed to take zero time. Between two consecutive macro steps, the same amount of time Δt passes such that $t_i = t_0 + i \cdot \Delta t$. Each macro step consists of a finite amount of zero, one, or more micro steps that are considered to be system-internal and unobservable from the outside; they are intermediate steps that the system performs to update its state.

Macro Step Phases. During the macro steps, all of the plants and controllers contained in the model execute their behavior in a sequence of micro steps. All plants are assumed to execute their behaviors first, followed by the execution of the controller behaviors. Macro steps are therefore not only subdivided into finite sequences of micro steps, but also into two phases as illustrated by Figure 3.14: Plant execution and controller execution. Conceptually, this two-phase execution approach allows the controllers to immediately react to changes in their plants' states: At some time t_n, the plants change their state in zero time through a sequence of micro steps. Within the same macro step, the controllers observe these changes through their sensors, compute the appropriate control actions, and update their actuators, all in zero time as well by performing multiple micro steps. Once all controllers are done, the macro step ends, time passes, and a new macro step t_{n+1} begins where the plants are influenced by the previous changes made to the actuators, causing plant state changes and completing the feedback loop. Sensors therefore observe the most recent plant states, whereas actuator effects are deferred to the following step to break up the feedback loop.

Feedback Loops. The model of the railroad crossing case study, for instance, contains several feedback loops as indicated by Figure 3.7a on page 30 that are broken up by this two-phase micro-macro steps execution approach: When a new macro step starts, the train advances its position, taking a potential emergency brake into account. Simultaneously, the barriers open or close, if requested, while the radio channel forwards

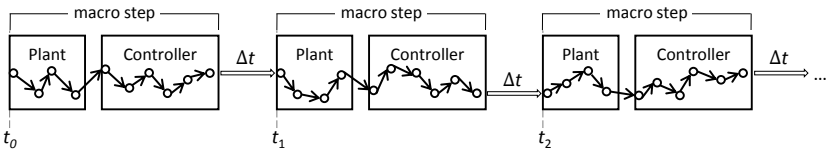


Figure 3.14. In addition to the partitioning into multiple micro steps, each macro step also separates plant behavior from controller behavior, with the plant behavior always executed first. The controller's last micro step ends the macro step, causing time to pass and a new macro step to begin; no time passes between the plant and controller parts of a macro step.

the messages sent between the two controllers. Subsequently, the train and crossing controllers observe their plants for relevant changes and react accordingly. For instance, the radio channel might forward the train controller's approach message to the crossing controller, causing the controller to activate the barrier motor in order to secure the crossing. All of these actions are micro steps belonging to the same macro step; starting with the next macro step, the barriers subsequently begin to close. Similarly, the larger feedback loop between the train and crossing controllers via the radio channel is broken up into at least two macro steps: When the train controller sends a message during macro step t_n, it is received by the crossing controller at t_{n+1}. If the latter decides to send an answer immediately, the train controller receives the response at t_{n+2}, i.e., the train controller must wait for at least two macro steps before any response can arrive.

Macro Step Execution. Each block in a system model can have two kinds of behavior: Behavior attached to the block's operations is executed whenever the corresponding operation is invoked, whereas the block's active behavior is executed autonomously during each macro step; such active behavior is based on the SysML concept of active blocks [153]. The model of computation places some restrictions on active behavior: The behavior must be cyclic and cannot terminate, each cycle is not allowed to span multiple macro steps, and it must start executing a new cycle when the block receives an Update signal. Each block is free to signal Update to other blocks, in most cases respecting the decomposition hierarchy while doing so. During a single macro step, the signal can be sent zero, one, or more times to the same block for maximum modeling flexibility. For example, a robot in the robot cell case study might never be triggered as long as it is currently not configured to perform any actions on any workpieces. On the other hand, multiple signals might be necessary for blocks that are required to reach a fix point during a macro step. Algorithm 1 illustrates the two phase execution process of macro steps, showing that each top-level plant or controller block of the overall system model is signaled exactly once; these blocks can subsequently forward the signals to other blocks as necessary, taking care of the model-specific distribution.

Concurrency. There are two different kinds of concurrency: Synchronous concurrency and asynchronous, interleaved concurrency [13]. The former is assumed throughout this thesis as it subsumes asynchronous concurrency, that is, interleaving executions of different components can be simulated in a synchronous context, albeit with sacrifices in analysis efficiency. In general, synchronous concurrency is more adequate for models of hardware, as different parts of a chip usually run in lockstep. Asynchronous concurrency, by contrast, is beneficial for interleaving software processes that are running on one

```
function MacroStep(plants : Set<Block>, controllers : Set<Block>)
    for p ∈ plants do
        Signal(p, Update)
    end for
    for c ∈ controllers do
        Signal(c, Update)
    end for
end function
```

Algorithm 1. An algorithm conceptually describing linearized macro step execution in two phases: Update signals are first sent in some arbitrary order to all plants before all controllers are signaled. All signals are sent sequentially, so there is no concurrent execution.

or more CPUs or that are distributed over multiple, mostly independent systems [164]. For safety-critical systems, asynchronous concurrency is inadequate, however, as arbitrarily interleaving the execution of plant and controller components would result in many unrealistic scenarios, making formal safety analysis mostly meaningless: Model checkers would analyze all situations where plants execute an arbitrary amount of steps before the controllers are allowed to react, which is completely unrealistic and violates the assumption of zero execution time. In fact, plant and controller execution are implicitly synchronized by time, which is more adequately captured by synchronous concurrency. Asynchronous concurrency, by contrast, is often encountered between different controllers such as the train and crossing controllers of the railroad case study. These two controllers are located in two physically separated locations and therefore execute independently in parallel, only synchronized by radio-based communication.

Linearization. Two blocks are considered to execute synchronously concurrent when their actions have no effect on each other during the same macro step. During analyses, the actual execution of these blocks can thus be linearized: If there are multiple plants or controllers, they execute their behavior sequentially in some unspecified order during the appropriate phases of the macro steps as shown by Algorithm 1; the resulting flow of Update signals through a system model is conceptually illustrated by Figure 3.15. Overall, linearization reduces neither modeling flexibility nor adequacy as both synchronous and asynchronous concurrent execution can still be explicitly modeled despite sequential execution during analyses. At the same time, analysis techniques do not have to consider concurrency explicitly, which can make them more efficient. Due to linearization, models can never assume a specific execution order of different blocks, but it is still possible for a model to force a specific order if desired: The blocks that should be explicitly ordered must be grouped together into one common parent block that schedules their executions appropriately. Algorithm 1 then sends the Update signal to this parent block at some point during macro step execution, which subsequently coordinates the executions of the original blocks by sending them Update signals whenever doing so is appropriate.

3.3.3 Modeling Component Behavior

SysML sequence diagrams, parametric diagrams, and state machine diagrams model behavior based on control flow, state equations, or state machines, respectively [153].

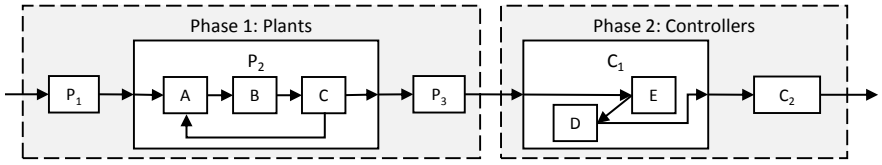


Figure 3.15. Illustration of the flow of Update signals (indicated by the arrows) through a model within a macro step: The plants receive the signal first, in turn signaling their subcomponents if necessary. During the second phase of the macro step, the controllers process the Update signal, also orchestrating their subcomponents, if any. The top-level blocks of each phase, i.e., P_1, P_2, and P_3 as well as C_1 and C_2, conceptually execute in parallel in some arbitrary order as determined by Algorithm 1. The signaling order of A, B, and C as well as D, and E, by contrast, is explicitly modeled by the parent blocks P_2 and C_1, respectively. In each macro step, the loop from C to A can only be taken finitely often as the macro step would not terminate otherwise.

For each behavioral aspect of a block, if there is more than one, the appropriate diagrams can be combined together to describe the block's overall behavior.

Modeling Control Flow. Figure 3.16 uses a SysML sequence diagram to give an overview of the distribution of Update signals to the components of the pressure tank case study. Due to the simplicity of this case study, the overall Update signal distribution behavior only consists of a simple sequence of Update notifications; more complex systems would make use of conditionals, loops, parallelism, and potentially nondeterminism. The ability to model control flow-based behavior is redundant when state machine-based behavior is supported by a modeling formalism but nevertheless useful for reasons of conciseness: Control flow consists of sequences of variable updates, conditional statements, loops, and so on, the entirety of which denote a control flow graph that could alternatively be expressed in a semantically equivalent state machine; conversely, there is always a control flow-based representation of any state machine. In most cases, however, one of the two forms of behavioral modeling is more natural than the other and increases model comprehensibility.

Modeling State Machines. State machines model state-dependent reactions to incoming events. Events can be receptions of Update signals or invocations of a provided operation of the block the state machine belongs to. The SysML syntax event [guard] / action denotes the event that triggers a transition, the guard that must hold for the transition to be eligible for being taken, as well as the control flow-based action that is executed when the transition is indeed taken. If multiple transitions are eligible, i.e., the current state has multiple outgoing transitions for the current event whose guards hold, one transition is chosen nondeterministically. Even though SysML leaves it undefined whether a transition takes time [154], the following always considers transitions to be instantaneous in accordance with the assumption of zero execution time. A state change therefore is a micro step, thus multiple transitions can be taken within the same macro step. In particular, the run-to-completion semantics of SysML state machines [121, 154] selects a maximally consistent set of enabled transitions whenever an event is triggered, potentially causing multiple transitions to be taken before any new incoming events are considered. State machines store a queue of incoming events that must be empty at the end of each macro step in order to facilitate model checking-based analyses.

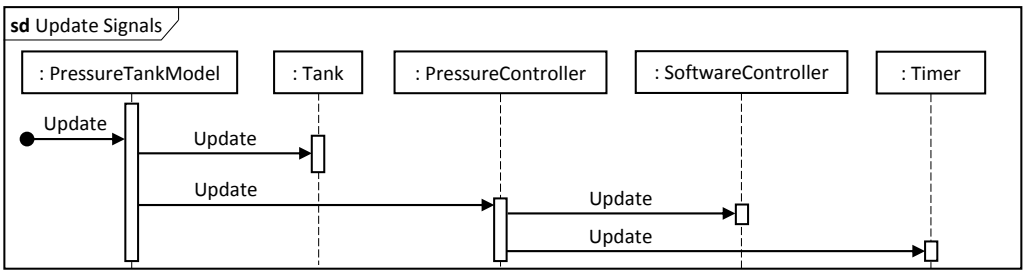


Figure 3.16. A SysML sequence diagram showing how Update signals are distributed through the model of the pressure tank case study. No signals are sent to the pressure sensor and the pump as they have no active behavior; they would simply ignore any Update signal sent to them.

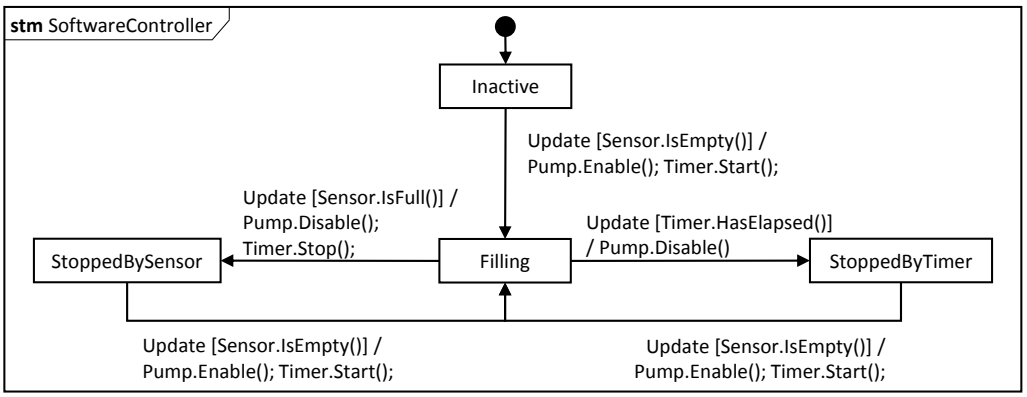
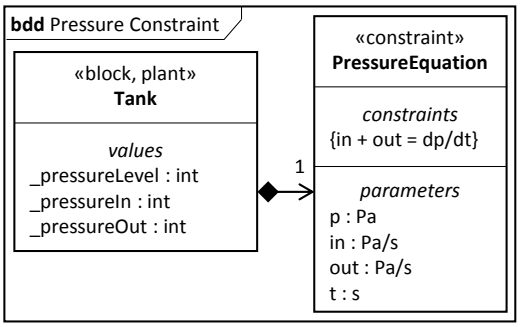
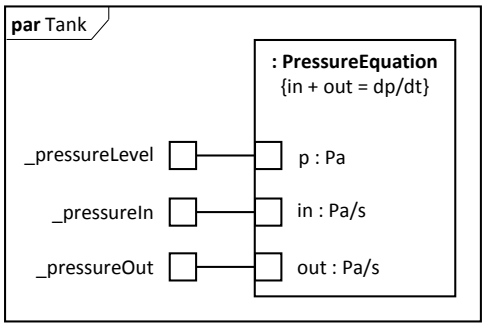


Figure 3.17. The SysML state machine representing the active behavior of the pressure tank case study's software controller: Initially, the controller is in the Inactive state. It switches to Filling as soon as the sensor reports that the tank is empty, activating the pump and starting the timeout. Filling continues until either the sensor reports that the tank is full or the timer signals a timeout, causing the controller to switch to its StoppedBySensor or StoppedByTimer states, respectively. The controller remains in these states until the tank is reported to be empty again.



(a) A SysML block definition diagram that adds the PressureEquation constraint to the Tank block declared in Figure 3.9 on page 35.



(b) A SysML parametric diagram that establishes the bindings between the Tank's values and the PressureEquation constraint's parameters.

Figure 3.18. The SysML specification of the relationship between the Tank's _pressureIn, _pressureOut, and _pressureLevel values using a constraint block and a parametric diagram. The time parameter t is assumed to be implicitly bound to a clock that is advanced with each macro step.

Each state machine attached to a block implicitly introduces additional state information, namely, a reference to the state the state machine is currently in. More advanced SysML features such as history states further increase the overall state information of the corresponding block, but are not considered in the remainder of this thesis. In the pressure tank case study, the state machine in Figure 3.17 represents the overall state information of the SoftwareController block from Figure 3.9, page 35, in its entirety, as the block does not declare any values or associations that change during analysis. The state machine in Figure 3.17 models the control software's active behavior: Whenever the controller receives an Update signal, it checks whether it has to change its state, executing the action associated with the chosen transition. For example, the state machine leaves the Inactive state when an Update signal is received and the pressure sensor reports the tank as being empty. Subsequently, the controller takes the transition, which in turn enables the pump, starts the timer, and changes the state machine's state to Filling. The state machine remains in this state at least until the next Update signal is received.

Controllers typically have an implicit or explicit model of their plant. For the pressure tank case study's control software, the state machine in Figure 3.17 represents the controller's implicit model of the tank: As long as the tank is being filled, i.e., the state machine is in state Filling, the tank is assumed to be non-ruptured and only partially filled. Once the controller switches to states StoppedBySensor or StoppedByTimer, it assumes the tank to be full. Subsequently, the pressure within the tank decreases, and the controller remains in either of the two states until the sensor reports the tank to be empty again. The controller's knowledge about the tank's actual state is therefore rather limited: There are only two situations in which it can be reasonably sure about the tank's pressure level, namely when the sensor reports the tank to be full or empty or when the timer signals a timeout. In all other cases, the controller neither knows nor cares about the tank's actual state. Sensor and timer faults such as ¬is full, ¬is empty, and ¬timeout cause the controller's perceived state of the tank to deviate from the actual state, potentially resulting in tank ruptures or complete tank depletions.

Modeling Equations. Figure 3.18 declares the equation that constrains the pressure level within the tank of the pressure tank case study using a SysML constraint block and a parametric diagram. The former represents a differential equation of the correlation between the tank's pressure and the amount of incoming and outgoing fluid, whereas the latter connects the equation's parameters to the tank's actual values. The tank's incoming and outgoing pressure is measured in Pascal per second; the outgoing pressure _pressureOut is assumed to be a negative constant to keep the case study simple, whereas the incoming pressure _pressureIn depends on the pump: If the pump is disabled, the incoming pressure is zero, otherwise it is some positive value. As illustrated by the sequence diagram in Figure 3.19, the tank changes the value of _pressureIn in response to receiving an Update signal by checking its IsBeingFilled required operation, which in turn is connected to the pump's IsEnabled provided port as previously shown in Figure 3.10 on page 37. The overall pressure delta $\mathrm{d}p/\mathrm{d}t$ for time t is thus equal to _pressureIn + _pressureOut as established by the PressureEquation constraint in Figure 3.18. For reasons of conciseness, the SysML model assumes continuous time, using a simple differential equation to describe the relationship between the pressure level as well as the incoming

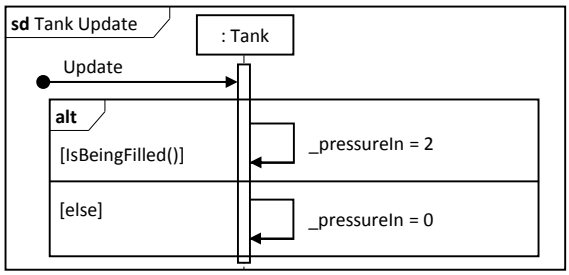


Figure 3.19. A sequence diagram showing how the tank updates its _pressureIn value in response to receiving an Update signal depending on whether the required operation IsBeingFilled returns true or false. As the tank is a plant block, it receives the Update signal before the control software. Consequently, the pressure tank controller always gets feedback about pressure level changes resulting from pump activations or deactivations in the previous macro step.

and outgoing pressures. Discretization of the relationship for time intervals Δt yields the discrete-state, discrete-time state formula $p_{n+1} = p_n + (\text{\_pressureIn} + \text{\_pressureOut}) \cdot \Delta t$.

3.4 Modeling Faults

Faults are a core concept in the context of safety analysis. They constitute an integral part of any model used for formal safety analyses even though they do not contribute to the system's actual functionality, i.e., to its nominal behavior. Instead, faults are modeling artifacts that are required to capture the system's fault behavior, subsequently allowing safety analysis techniques to reason about their impacts on the system's overall behavior as well as the situations leading to hazard occurrences. Consequently, the models contain both nominal and off-nominal system behavior, that is, the models not only describe the intended and desired system functionality but also the unintended behavior in the presence of one or more faults.

The common terms of faults, errors, and failures have been used consistently in the preceding sections using the terminology summarized by Avižienis et al. [9], but now require a more thorough explanation in order to establish the relationships between them as shown in Figure 3.20: Safety analyses consider situations in which faults cause off-nominal system behavior that would not have occurred otherwise. These situations represent fault activations, that is, a fault is activated when it influences the system through effects that change the internal state of affected components, thereby causing errors. Thus, errors are deviations of the components' states from what they should have been, whereas a fault is one of possibly many causes of an error. Errors propagate through the components, causing other errors. Eventually, errors might result in failures where the errors manifest themselves in a way that is externally observable. Failures therefore represent observable errors, i.e., situations in which a component evidently does not behave as specified or expected. Failures either cause faults in other components or they represent hazards, that is, safety-critical failures at system level. Safety analyses are conducted for the latter to determine all faults causing the hazards by uncovering the cause and consequence relationships between such system-level failures and the individual faults contained in the system model.

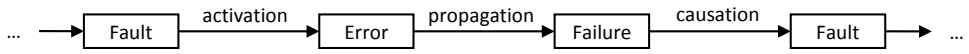


Figure 3.20. Overview of the relationship between faults, error, and failures: Activations of faults result in errors, which propagate through the system until they become externally observable failures. Failures, in turn, can cause other faults, continuing the propagation chain [9].

Faults. Faults represent causes for errors. There are three main, partially overlapping classes of faults that characterize the fault origins [9]: Development, physical, and interaction faults. Development faults are made during development, the most prominent ones being software bugs, that, when activated, result in errors due to incorrect computations. Physical faults affect hardware, resulting in hardware failures such as ¬pumping in the pressure tank case study. Interaction faults are external to the system, i.e., they originate from unexpected environmental disturbances such as flood waves of ocean water or human-made command faults like physical manipulations of the pressure tank that destabilize the tank's structural integrity, increasing the likelihood of tank ruptures. Omission faults and commission faults, in particular, represent the absence of required human commands or wrong commands, respectively. A command fault might be deliberate to prevent other, even more catastrophic consequences, while malicious command faults are intended to provoke exploits that allow unauthorized access to or modifications of the system.

Fault Persistency. Faults that are not currently causing an error are dormant until they are activated and become active, turning dormant again when they are deactivated. Activations are either triggered from within the system, e.g., by executing certain computations, or by external environmental conditions. A fault's persistency constrains the possible transitions between the active and dormant states as illustrated by Figure 3.21. In the railroad crossing case study, for instance, failures of the train's brakes are most adequately modeled as permanent faults: Once they are broken, they cannot magically repair themselves and start functioning again. If maintenance intervals are to be considered in the model, a hybrid kind of persistency could be used for brake failures where the fault is permanent until the brakes are repaired, causing the fault to become dormant again. On the other hand, communication messages could be lost because of atmospheric disturbances, e.g., thunderstorms, or large objects such as mountains blocking the transmission depending on the exact positions the messages are sent from, for which the exact occurrences cannot be accurately predicted. Such faults are thus best modeled with transient persistency to capture all possibilities without missing any, which could otherwise invalidate the adequacy of the safety analyses. Transient faults represent the general case, subsuming all other kinds of fault persistency, including permanent faults. It is therefore always possible to assume transient persistency without risking any safety analysis oversights. Thus, a fault that is permanent in reality but modeled as transient does not invalidate safety analysis results, even though the system might seem less safe than it actually is.

Errors. Errors are the results of fault activations and their effects on the affected components, representing deviations of component states that might eventually lead to failures. In the pressure tank case study, for example, an activation of the ¬pumping fault

(a) Transient faults switch between their active and dormant states completely nondeterministically.

(b) Permanent faults are activated nondeterministically, but can never reach their dormant state again.

Figure 3.21. SysML state machines describing the transitions of transient and permanent faults between their active and dormant states. These two kinds of persistency are the most common ones, but other kinds also exist.

causes the pump to stop even though it should continue to run; the error caused by the fault is the discrepancy between the pump's actual running state and the expected one. In this case, the fault activation results in an error that also immediately represents a failure of the affected component, whereas in general, some time can pass before an error becomes externally observable. Not all fault activations must result in failures: On the one hand, the system could protect itself against individual errors using redundancy, for instance. On the other hand, some errors never result in failures because, for instance, they occur in some inactive, unused part of the system; alternatively, an error could be eliminated unintentionally before it has the chance to cause a failure, e.g., an erroneous value might be overwritten by a valid one before it is used in a computation. While there can be many faults that result in the same error, there can also be many errors caused by the same fault, in particular in different component instances. Such related errors are the results of common cause faults; e.g., a tsunami set off the sequence of events that lead to the Fukushima Daiichi nuclear disaster by destroying the emergency power generators, making reactor cooling impossible [100].

Failures. A failure occurs when a component or the entire system observably deviates from its intended and expected behavior; the different ways that failures can manifest themselves are called failure modes. Typical failure modes include incorrect or missing data as well as actions undertaken at the wrong time, i.e., either too early or too late [9, 131]. For instance, both the ¬is full and ¬is empty faults lead to a failure of the pressure sensor in the pressure tank case study: Either the sensor reports a full tank as non-full, or it reports an empty tank as non-empty. In both cases, the sensor does not behave as specified and hence fails. The failure modes, by contrast, describe how the sensor deviates from its expected behavior, that is, by either reporting a full or an empty tank incorrectly. At the level of abstraction that the pressure tank case study is analyzed in, the pressure senor's faults and failure modes overlap and there are no intermediate errors caused by the faults. If the sensor was modeled and analyzed in more detail, however, its internal errors would become apparent, uncovering the chain of fault activations, error propagations, and eventual failures.

Relationship between Faults, Errors, and Failures. Figure 3.22 gives a more concrete overview of the propagation of faults, errors, and failures than Figure 3.20, showing that the failure of a component can cause faults in other components that depend on it. Due to this fundamental chain of error propagation, the faults leading to a hazard might

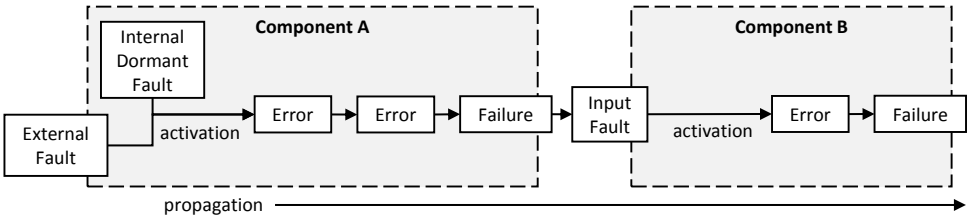


Figure 3.22. Illustration of error propagation between two components [9]: Either a fault external to component A or an internal, but dormant fault causes an error within A upon activation. Through a series of state changes or computations, the error propagates through the component until it eventually reaches the component's boundary. At the boundary, the error becomes externally observable, thus representing a failure. Due to an assumed dependency of component B on A, A's failure causes an input fault in B, which B is likely to be completely unaware of. Error propagation continues, eventually causing a failure of B.

lie in various different parts of a system that might seem completely unrelated. Moreover, faults can be activated a long time before the system actually fails. A non-exhaustive overview of potential faults affecting the different parts of a safety-critical system is shown in Figure 3.23: Sensor faults often cause errors in the controller's internal model of its plant that remain undetected for some time until a hazard occurs or the controller is able to correct them through feedback or luck. In the pressure tank case study, the ¬is full fault causes an error in the controller's perceived plant state, which is corrected during the next system step through the timer signaling a timeout. A failure of the pump caused by the ¬pumping fault remains undetected until the sensor continuously reports the tank to be empty. In general, the same plant could also be affected by multiple controllers that could issue conflicting control actions due to synchronization issues or inadvertently due to faults. For example, the controller that handles the removal of fluid from the pressure tank could potentially also pump fluid into the tank instead of removing fluid due to a fault reversing the flow of the fluid. Such a situation is also likely to result in a tank rupture.

Safety versus Functional Correctness. Verification of functional correctness is primarily concerned with the control algorithms, the controllers' plant models, and updates of the plant models at run time in accordance with the plants' actual behavior. That is, functional correctness deals only with the development faults listed within the controller in Figure 3.23. Additionally, however, inconsistent, incorrect, or incomplete plant models can also be errors resulting from plant or sensor faults. Development faults such as software bugs or system design issues can be fixed once discovered during functional correctness verification, preventing them from causing errors during system operation. Physical faults as well as interaction faults, by contrast, cannot be prevented by their very nature; the system can only be designed with error detection and prevention mechanisms such as self-organization that work around the errors caused by the activations of the faults. For model-based safety analysis, it is thus important to consider all of these potential sources of faults, additionally requiring the controlled plants to be modeled as well. Consequently, formal safety analyses subsume functional correctness verification. However, it is usually already challenging to create adequate models of

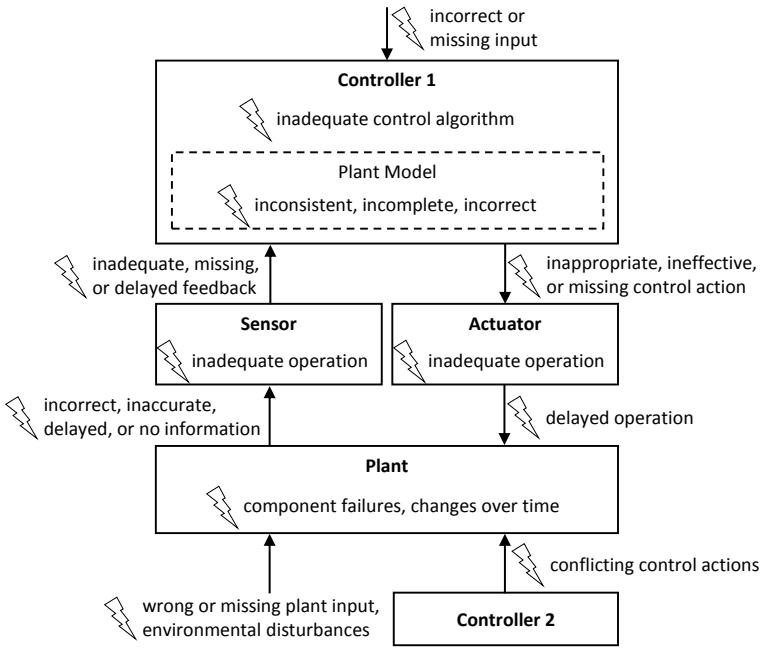


Figure 3.23. Overview of the basic structure of safety-critical systems, annotated with potential sources of faults and failure modes [131]. Faults are either system-internal or -external, occurring either in some system component or outside of the system's boundaries. Incorrect or missing input that the controller depends on can result in hazards just like environmental disturbances affecting the plant. Multiple controllers can issue conflicting control actions, while control actions themselves can be inappropriate, ineffective, delayed, or missing due to actuator faults. Sensors provide feedback to the controller, allowing it to measure the effects of its control actions on the system. Feedback is critical for safe system operation, and any incorrect, inaccurate, inadequate, delayed, or missing feedback due to sensor faults increases the likelihood of hazards. The plant can also change over time, for example due to wearout or modifications during maintenance, or its input from other plants might be missing or incorrect.

a system's nominal behavior. The system's off-nominal behavior in case of faults is often even harder to predict and model correctly and adequately. Model checking-based safety analysis techniques help by making it possible to model the local effects that a fault has on the affected components only; the model checker is able to uncover the chain of error propagations automatically in order to the determine the global impacts of the faults on the system's overall behavior.

3.4.1 Fault Modeling Guidelines

The following four fault modeling guidelines encourage a strict separation of concerns [33] regarding various fault modeling activities. As reported by Joshi et al. [113], manual integration of off-nominal behavior into the model of a system's nominal behavior quickly results in cluttered models in which the nominal behavior is hard to keep track of. For this reason, it is beneficial to specify nominal and off-nominal behavior separately, letting a tool generate the combined model automatically [26, 27, 135]. The

fault modeling guidelines therefore focus on basic principles on how to best describe faults and their effects, leaving the actual integration of both kinds of behaviors to the modeling languages and the analysis tools.

Guideline 8 (Separation of Nominal and Off-Nominal Behavior)**.** *Nominal and off-nominal system behavior should be clearly separated, allowing later extensions of the model with additional off-nominal behavior as well as automated removal of off-nominal behavior. The integration of both kinds of behavior should be automatic.*

The separation of the models into nominal parts and off-nominal parts supports subsequent modeling and analysis tasks [113]: For example, it must be possible to isolate the nominal behavior contained within a model for verification of functional correctness independently from the off-nominal behavior. Additionally, automated test case generation for system functionality or code generation can only be supported if the off-nominal parts of the model can be identified, as obviously no code should be generated for off-nominal behavior, for instance.

Guideline 9 (Separation of Effects and Persistency)**.** *The persistency of a fault and its effects, i.e., the ways in which the fault causes errors, are orthogonal concerns and should therefore be modeled separately.*

Fault modeling consist of two orthogonal parts [160]: Faults have effects that describe the affected component instances as well as the errors caused by the faults' activations, whereas the faults' persistencies restrict their transitions between their active and dormant states. Fault effect modeling is the primary concern; for fault persistency, transient persistency is often a good initial guess as transient faults are the general case. If the assumption is invalid, it results in extremely unlikely or physically impossible situations that are discovered during safety analyses. Due to the separation of effects and persistency, however, a fault's persistency can easily be changed without affecting any other parts of the system model.

Guideline 10 (Effect Localization)**.** *Fault effects should be localized to the components that are directly affected by the faults; in particular, error propagation chains and input faults should not be modeled explicitly.*

Fault identification and modeling is challenging due to the large variety of possible faults and effects. Explicitly modeling the chains of error propagation, which are often not particularly obvious, would add even more complexity to the model, increasing the danger of fatal oversights that invalidate formal safety analysis results. On the other hand, safety analysis tools such as HiP-HOPS that require explicit error propagation models [181] are faster than model checking-based analysis techniques that can automatically deduce error propagation sequences. That is, modeling effort reductions due to implicit error propagation come with the cost of increased analysis effort, which often is an acceptable tradeoff considering the improvements in safety analysis accuracy that are achieved by uncovering propagation chains automatically.

Guideline 11 (Effect Compositionality)**.** *Fault effects should be modeled compositionally, allowing the automated application of multiple fault effects on the same component instances with optional prioritization.*

As there usually are multiple faults affecting a component, the faults' effects potentially cause errors in the same parts of the component's state. It must not only be possible to model such faults, but also to analyze combined activations of such conflicting faults. In the height control case study, for example, there are two faults affecting the light barriers: ¬detection and ↯detection both affect the same provided operation IsVehicleDetected. Either of the faults could take priority, one of the effects could be chosen nondeterministically, or some computation could be run to combine the two effects. In this example, nondeterminism seems to be the best choice, as no further assumptions are made about the inner workings of a light barrier that would justify a prioritization. However, it is still possible to prioritize either of the faults to improve model checking efficiency in this case: As the IsVehicleDetected operation only returns a Boolean value, the nondeterministic choice does not result in any situation that has not already been discovered by analyzing each fault individually, hence in this case, combined activations of both faults are irrelevant.

3.4.2 Fault Injection

The nominal behavior of a safety-critical system is commonly modeled first with the off-nominal behavior added later in a separate step to cope with modeling complexity [5, 9, 24, 65, 80]. The term fault injection is used to describe the process of extending the nominal behavior with off-nominal behavior in the presence of one or more faults, resulting in an extended model of the system [24]. Faults are injected into the model in order to allow formal safety analysis techniques to reason about off-nominal behavior, facilitating the automatic computation of fault-related causes that lead to a hazard. In this sense, the meaning of the term fault injection in the context of this thesis is slightly different than the one sometimes found in literature, where faults are injected into a model in order to test and validate the validation techniques themselves, i.e., to ascertain that all injected faults can be found by the verification and validation techniques used to analyze the system [5]. For formal safety analysis, faults are not injected to validate the techniques, but to enable the analyses in the first place.

Fault injection is a purely additive process, as it merely extends the nominal system behavior: As long as no faults are activated, the extended model is behaviorally identical to the original one, thus, the off-nominal model is a conservative extension of the nominal one [80]. This restriction is irrelevant for the soundness and completeness of formal safety analysis techniques, but it is an important consideration for model adequacy. If the process of extending the nominal model itself ever removed nominal behavior, it would simply indicate that the faults are modeled inadequately or that the fault injection mechanism is incorrect. It is of course possible, and very common, that activations of some faults prevent previously possible nominal behavior, either removing it entirely or replacing it with other, off-nominal behavior. A fault preventing nominal behavior captures the intended effects of the fault incorporated into the model, whereas behavior suppression due to the fault injection process itself is simply a modeling anomaly that must be avoided. For this reason, S# and many other safety analysis tools support adequate fault modeling by guaranteeing correct model extension either syntactically or at least semantically [27, 82].

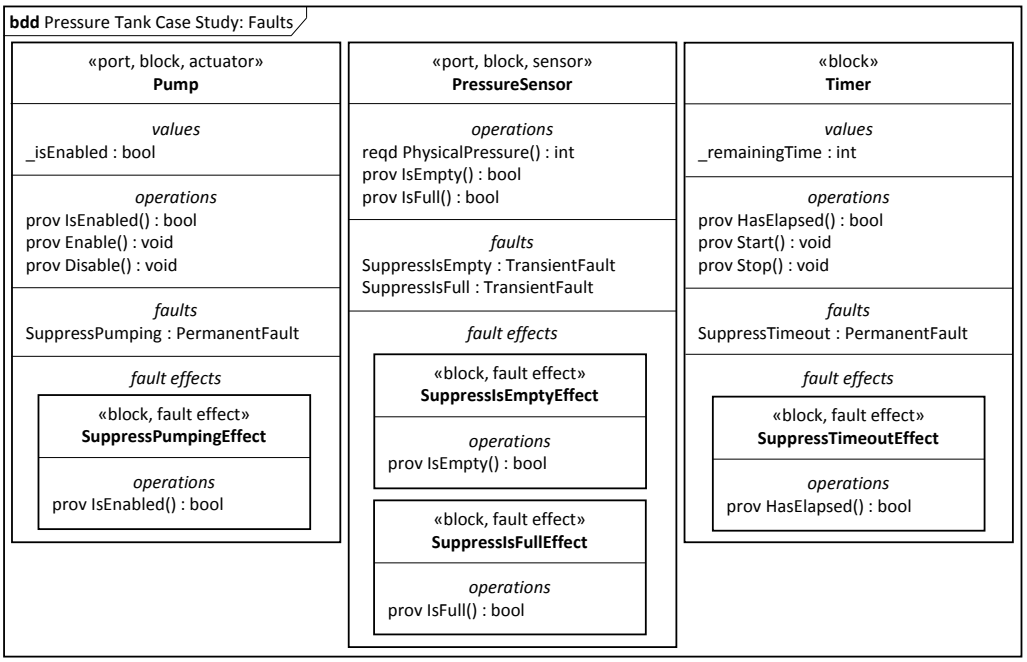


Figure 3.24. A SysML block definition diagram showing some of the blocks from Figure 3.9 on page 35 with faults and fault effects. Faults are instances of a Fault-derived type such as TransientFault or PermanentFault that determine their persistency. Fault effects are nested blocks marked with the «fault effect» stereotype, overriding the behaviors associated with one or more required or provided operations of their containing blocks. The overridden behavior is executed instead of the original one when the fault corresponding to an effect is activated, either leading to latent errors within the component or to externally observable failures.

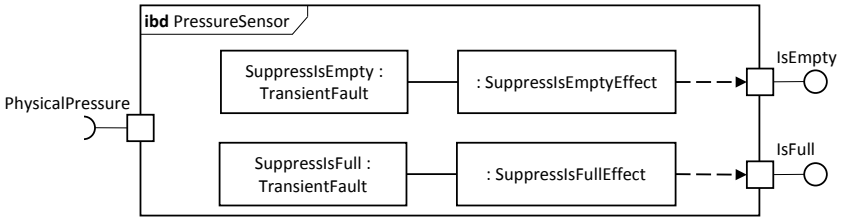


Figure 3.25. A SysML internal block diagram of the PressureSensor block showing the association between its Fault instances and instances of its «fault effect» blocks. The dashed arrows indicate which operations are affected by a fault effect.

Figures 3.24 and 3.25 show some of the parts of the SysML model for the pressure tank case study where the four faults ¬is full, ¬is empty, ¬pumping, and ¬timeout are injected under the names SuppressIsFull, SuppressIsEmpty, SuppressPumping, and SuppressTimeout, respectively. Faults are modeled as properties of the blocks they affect, though they could also be declared in other blocks in case of common cause faults affecting multiple component instances. Fault effects are modeled as nested blocks, clearly denoted by the «fault effect» stereotype, indicating the operations of the containing block that they affect. Blocks marked with the «fault effect» stereotype are also allowed to override the

containing block's active behavior, changing how the block reacts to receptions of the Update signal. There are no limits imposed on what a fault effect is allowed to do: It can introduce additional state, change the return value of the operation it overrides causing immediate failures, or change some of the containing block's values or associations, causing latent errors.

The extended model of the pressure tank case study shown in Figures 3.24 and 3.25 clearly separates the nominal and off-nominal behavior of a block in accordance with Guideline 8. Additionally, the partitioning into Fault instances and «fault effect» blocks satisfies Guideline 9, with internal block diagrams such as the one shown in Figure 3.25 linking the two parts together. The two fault effects affecting the pressure sensor demonstrate that fault effect modeling is compositional as required by Guideline 11, while the nesting of «fault effect» blocks adheres to Guideline 10 by restricting a fault's effects to the component it influences directly, instead of explicitly modeling error propagations.

3.5 Related Work

Expressiveness of Modeling Languages. Most model checkers such as NuSMV, SPIN, LTSmin, Uppaal, Prism, among others, have very low-level specification languages whose level of expressiveness is only marginally higher than that of Kripke structures or Markov decision processes in the probabilistic case [17, 38, 98, 116, 123]. Adequate modeling of complex safety-critical systems directly in the input languages of these model checkers is therefore quite involved and time-consuming, especially if design variants of the systems are also to be modeled and analyzed. It is thus beneficial to use higher-level modeling languages, resorting to a compiler that performs automated model transformations into the input languages of the target model checkers [195]. This approach is followed by most tools for formal safety analysis, including VECS, HiP-HOPS, Compass, AltaRica, and others [15, 27, 135, 181]. In contrast, the approach followed in this thesis is completely different: Instead of relying on model transformations, the S# high-level models, authored in a textual representation of an executable, extended subset of SysML, are executed during model checking, conceptually generating the low-level Kripke structures on-the-fly during model execution with the help of the explicit-state LTSmin model checker as explained in Chapter 6. Compared to model transformations, S#'s model execution approach is simpler to implement and has no potential issues with semantic consistency between the high-level and low-level models; still, S#'s analysis efficiency is in general on par with that of other safety analysis tools, cf. Section 6.5.

Structural Modeling Paradigms. What most safety analysis tools have in common is their component-oriented approach to system structuring, most likely due to the fact that system decomposition into an assembly of interconnected components is the most natural way of thinking about safety-critical systems. The Compass toolset, for instance, is based on the SAE standardized architecture analysis and design language (AADL) [60, 151, 183] which uses components as its primary structuring mechanism. AltaRica and VECS define custom languages around components, even though the latter does not enforce strong encapsulation of component state [15, 135]. While AADL even requires explicit interface modeling for all components and therefore supports

flexible model composition, AltaRica is somewhat less flexible in this regard. VECS, by contrast, sits in between the two and allows for convenient modeling of functional design alternatives while structural variants are less well-supported due to its weaker encapsulation of individual components. Scade, AutoFocus, Quartz, HiP-HOPS, and Ptolemy are other examples for tools and approaches using custom, component-oriented modeling languages for safety-critical systems [2, 97, 173, 179, 181], whereas object-oriented ones exist as well, cf. Modelica, ABS, SystemC, or the HATS project [40, 73, 112, 145]. SysML and UML, on the other hand, can be used in both component-oriented and object-oriented ways, bridging the gap between both approaches and allowing models to use the paradigm that is the best fit for the modeled system. In particular, it is often convenient to model the overall system structure with components, whereas more detailed models of controller components sometimes benefit from a more object-oriented approach as can be seen for the robot cell case study in Chapter 8.

SysML- and UML-Based Approaches. SysML- and UML-based safety modeling and analysis approaches generally use the same models for both safety analysis and general system modeling, avoiding any discrepancies that could potentially arise if different models are created manually [142]. Helle [91], for example, uses combinations of SysML block definition diagrams, internal block diagrams, activity diagrams, and use case diagrams to describe the nominal and off-nominal behavior of a safety-critical system; subsequently, it is possible to automatically extract the minimal critical fault sets from the model. Alternatively, Mhenni et al. [143] use known modeling patterns in the construction of internal block diagrams to facilitate automatic construction of fault trees or FMEA tables [144]. SOPHIA and QuantUM [35, 129], on the other hand, define UML profiles to model quantitative aspects of safety-critical systems, directly calculating tolerable accident rates or using Prism to compute hazard probabilities, respectively. The latter maps Prism counterexamples back to UML sequence diagrams, allowing the counterexamples to be evaluated in the context of the high-level model to improve their comprehensibility [195]. All of these approaches require explicit error propagation models, thereby reducing analysis times at the expense of additional modeling effort and an increased risk of modeling oversights.

Modeling Philosophy. Compared to the modeling approach outlined in this chapter, the aforementioned tools and techniques have stricter requirements about the way the systems have to be modeled, which, to a certain extent, limits the expressiveness provided by SysML that can be used. Such limits can be a good thing and can indeed help to improve adequacy, but they might be too strict for some systems where they require workarounds that decrease model comprehensibility. While the guidelines in the preceding sections encode fundamental suggestions on how to model system structure, behavior, and faults, they are indeed only suggestions that, with caution, can be purposely violated if necessary and justified. For example, the VECS tool has stricter guidelines encoded into the modeling language that, for example, do not allow references to other components to be stored [135, 188], resulting in a fixed hierarchical component composition. In particular, self-organizing systems require more modeling flexibility to cope with the inherent complexity of these systems as discussed in Chapter 8. This difference between both points of view is a matter of design philosophy: The guidelines

could be strictly enforced to avoid accidental mistakes, which might be acceptable or even beneficial in many cases but comes with the cost of negatively impacting the remaining ones in a more or less profound way. Therefore, the guidelines only try to steer the models towards "the pit of success" [8], meaning that the set of guidelines encourages the creation of correct and adequate models without compromising expressiveness in any way. This thesis generally follows the latter philosophy, which is also the governing design principle [136] for the C# programming language and the .NET framework that the executable modeling approach presented in the next chapter is based on.

Abstraction of Timing Behavior. The assumption of zero execution time abstracts the notion of time in the system models, assuming that zero time passes during the computation of a system's reaction to its input stimuli. Lustre, Esterel, and Quartz are well-known examples of programming languages for embedded systems that are based on the notion of zero execution time [86, 179, 189]. Before the invention of these languages, physical or bounded execution time were the norm [119] and developers had to manually consider worst case execution times and process scheduling algorithms. Logical execution time is another alternative for abstract time modeling [119], supported, for instance, by the Giotto programming language for embedded control systems [92]. Compared to zero execution time, timing behavior of the system is considered in more detail, allowing the real time scheduling algorithms of the embedded operating system running the software to ensure correct timing behavior. As models of safety-critical systems are typically created with a higher level of abstraction than the actual implementations of their controller software, zero execution time is a reasonable assumption to make compared to the aforementioned alternatives [20]. Recent developments also try to relax the assumption of overall zero execution time to increase modeling flexibility, allowing less safety- or mission-critical parts of a system to actually take time [207].

Model of Computation. The model of computation presented in this chapter employs macro and micro steps to capture the notion of zero execution time, similar to synchronous languages such as Esterel or Lustre [20, 86, 189] or programming interfaces for robotics applications [196]. Alternatively, a delta cycle semantics could have been used similar to SystemC [73]. SystemC provides a set of C++ classes and a simulation environment for system-level modeling, software development, and hardware synthesis based on the tooling and language features provided by the C++ programming language. SystemC's delta cycle semantics is based on separate evaluation and update phases to allow for inter-process communication and coordination within the same cycle. Time only advances when a fix point is reached, i.e., no system components issue any delta notifications during the evaluation phase, allowing the update phase to advance time. In a sense, SystemC therefore also differentiates between macro and micro steps, using more complex mechanisms to determine the end of a macro step in an attempt to automatically break up feedback cycles between different system components. In doing so, it requires queues to track the components that have to be schedule next which can become problematic for model-based analysis techniques due to state space explosion. The overall complexity of the delta cycle semantics thus makes it hard to model check SystemC models efficiently even though some model checking techniques based on transformations to the SPIN model checker exist [39], for example.

Hybrid, Timed, and Probabilistic Systems. This thesis considers discrete-state, discrete-time models only, requiring manual discretization of continuous behavior. Additionally, the models do not contain any information about fault probabilities, enabling only qualitative safety analyses that cannot compute hazard probabilities. Traditional techniques for probability estimation [80, 156, 193] must be used instead to compute hazard probabilities from the formal analysis results by making some simplifying assumptions such as stochastical independence of the analyzed faults. Prism and MRMC [117, 123], on the other hand, are probabilistic model checkers that are used by the VECS, Compass, and QuantUM tools for more accurate but less efficient probabilistic safety analyses [129, 135, 151], for example. Some of these tools, such as the Compass toolset, even support timed or hybrid behavior, whereas Modelica has first-class support for system modeling with differential equations [27, 145]. While hybrid models consisting of both discrete-state and continuous-state parts as well as discrete-time and continuous-time parts can improve modeling adequacy, they are likely to complicate analysis techniques to the point where they become either completely infeasible or at least must resort to statistical, non-exhaustive analysis approaches [58, 118].

Error Propagation. Classical safety analysis techniques such as FTA and FMEA [166, 193] or more recently developed variants such as Component Fault Trees [74, 114] do not require any fault modeling, emphasizing the actual analyses instead of model creation. Model-based safety analysis techniques and tools, by contrast, play down the importance of the analysis processes, instead focusing on the modeling activities [25]. These latter tools fall into two broad categories: Error propagation is either implicit or explicit. Implicit error propagation to some extent always requires models of system behavior in order to enable the analysis tools to compute the missing information automatically, whereas explicit error propagation requires manual identification, understanding, and analysis of these interdependencies. VECS, Compass, FSAP/NuSMV, xSap, and S# fall into the former category with implicit error propagation [24, 65, 82, 135, 151], for instance, whereas HiP-HOPS, failure propagation analysis, QuantUM and other SysML- or UML-based approaches fall into the latter [70, 129, 162, 163].

Fault Modeling. Most model checkers provide no explicit support for fault modeling [38, 98, 116, 123], complicating adequate fault injection by not helping with the process of conservatively extending the nominal model into the off-nominal one [80]. Tools and techniques specifically developed for formal safety analysis, on the other hand, often guarantee conservative extension by construction or at least provide modeling guidelines that indicate which changes are permitted during model extension; e.g., Compass and S# fall into the former category, whereas VECS belongs to the latter [82, 135, 151]. In contrast to older tools such as FSAP/NuSMV [24], more recent ones such as VECS, Compass, xSAP, or S# all support flexible modeling of fault persistency [65, 82, 135, 151]. The biggest difference between these tools lies in the way fault effects are modeled: In VECS, actual component behavior must be changed; in Compass and xSAP, failure behavior is modeled in separate models that are linked together by a tool; and in this thesis, both nominal and off-nominal behavior is separately specified within different parts of the same model, that is, by additional «fault effect» blocks added to the components that are affected by the faults.

Summary and Outlook. The systematic modeling guidelines presented in this chapter build upon established system- and control-theoretical basics as well as software engineering principles and best practices: Safety-critical systems are decomposed into hierarchies of components that either represents sensors, controllers, actuators, or plants. The feedback loops between plants and controllers are important to model in order to enable complete safety analyses; if the plants were not taken into account, only functional correctness could be verified, in general. Additionally, the faults that can affect the systems to be analyzed must be injected into the system models, extending the nominal behavior with off-nominal behavior to allow safety analysis techniques to reason about chains of error propagations. Most of safety-specific tools and techniques emphasize a more or less systematic approach to fault modeling, but in general do not provide any guidelines on how to best model safety-critical systems overall. The systematic modeling guidelines presented in this chapter, however, are generally applicable and in no way limited to SysML or S#. Thus, these guidelines can also be followed when other modeling approaches are used, at least as long as the required modeling mechanisms for compositional modeling, among others, are supported.

The S# modeling and analysis framework for safety-critical systems that is introduced in Chapters 4 and 6 builds upon the modeling guidelines of this chapter. In particular, it closely follows the model of computation, allowing the creation of discrete-state, discrete-time models with a micro-macro step semantics. Furthermore, the S# modeling language supports the systematic fault modeling approach at the syntactic level already, clearly differentiating between faults, their persistencies, as well as their effects; fault activations and error propagations are implicitly handled during analyses. The formal techniques that form the foundation of safety analyses carried out by the S# framework are introduced in Chapter 5. These techniques are also rooted in the systematic fault modeling approach presented in this chapter, making the concept of fault activations central to the formal models through a fault-aware modeling and specification approach. Systematic modeling therefore not only helps to foster the creation of comprehensible and adequate models, it also supports the formalization of the underlying safety analysis techniques, for instance by facilitating the formal definition of fault injection and its dual, fault removal. Chapters 7 and 8 subsequently use the modeling and analysis features offered by S# to systematically model and analyze the height control case study as well as the self-organization case studies.

Summary. The S# modeling and analysis framework for safety-critical systems is based on the ISO-standardized C# programming language and the .NET run time environment [105, 110]. It is a textual representation of an extended subset of SysML, allowing the creation of executable models of safety-critical systems in a systematic and expressive way. Due to its C#-heritage, S# offers flexible design variant modeling and composition capabilities, while extensions of C# core concepts allow for systematic fault modeling [102]. Executability is S#'s main characteristic that facilitates model simulations, visualizations, as well as model checking-based analyses as explained in subsequent chapters.

Publications. The modeling language and design goals of S# are presented in [82, 84]. This introduction to the modeling language features is based on the S# Wiki [102].

4

Executable Models of Safety-Critical Systems

There are three aspects to consider for a systematic modeling approach: The syntax of the modeling language, its semantics, as well as the methodology supporting the approach. The latter typically consist of systematic modeling guidelines that are usually at least partially independent of the modeling languages and are therefore considered separately. In the preceding chapter, for instance, the high-level modeling language SysML is used to model safety-critical systems by following the systematic structural, behavioral, and fault-related modeling guidelines introduced for safety-critical systems; other modeling languages could make use of the same guidelines. However, while SysML is an expressive language that is well-established in the field of embedded systems development, including safety-critical ones [79], it is not designed for formal analyses: Its informal semantics leaves up multiple semantic variation points for individual interpretation [153, 154]. Additionally, it does not readily support model checking-based analyses or model simulations in order to be able to verify and validate whether the created models are correct and adequate.

The S# modeling and analysis framework for safety-critical systems is designed to work around SysML's shortcomings for formal safety analysis while retaining its strengths,

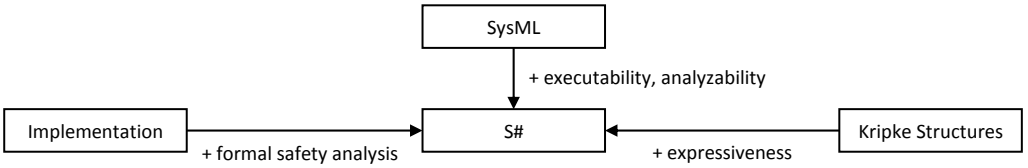


Figure 4.1. Illustration of the relationship between S#, SysML, Kripke structures, and actual implementations of safety-critical systems: S# sits between SysML and Kripke structures with a level of expressiveness similar to SysML but with the formal analyzability of Kripke structures. Additionally, S# bridges the gap between SysML modeling and actual implementation-level languages such as C++, allowing both model execution as well as formal safety analysis.

namely its high level of expressiveness. Figure 4.1 illustrates the gap [79, 85] that S# is designed to fill, sitting in between informal, high-level modeling languages such as SysML, low-level formalisms such as Kripke structures used for formal analyses, and the actual implementations of the safety-critical systems. Both SysML and S# are roughly comparable concerning modeling expressiveness, but only S# allows its models to be executed with fixed semantics. SysML can thus be seen as a graphical representation of S# models that can be used when a graphical notation of a modeling artifact seems more understandable than a textual one; ideally, there would be a tool for automatic two-way transformations between the two, however, no such tool exists yet at the time of writing. On the other hand, S# models are analyzable similar to Kripke structures using the unified model execution approach discussed in Chapter 6 but have a significantly higher level of expressiveness. Compared to the actual implementation of a safety-critical system, which obviously is also executable, S# models facilitate formal safety analyses. The abstraction levels differ between models and implementations as the former are typically not concerned with details such as sensor access APIs or low-level network protocols. Additionally, the models contain plant behavior that often does not have to be implemented such as the vehicles in the height control case study. Furthermore, off-nominal behavior is a modeling artifact required for formal safety analyses that is never implemented at all. Implementations can therefore be checked for functional correctness via tests, for instance, but only allow for limited safety assessments.

This chapter starts with an overview of the S# modeling and analysis framework in Section 4.1. Subsequently, Sections 4.2 to 4.5 give a basic introduction to S# modeling language features; more details are available in the S# Wiki [102]. S#'s unified execution approach is subsequently discussed in chapter Chapter 6 after the foundations of formal safety analysis are established in Chapter 5.

4.1 The S# Modeling and Analysis Framework

The S# framework brings forward well-established software engineering principles and best practices to the modeling and analysis of safety-critical systems during all phases of development. It is an integrated approach that fosters the systematic development of comprehensible, adequate, and modular models of safety-critical systems, while also allowing the use of efficient, formal safety analysis techniques for rigorous safety assess-

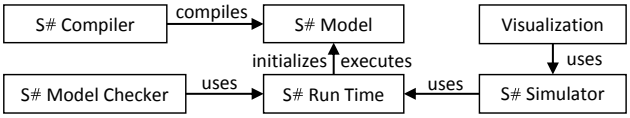


Figure 4.2. Illustration of S#'s execution-centric architecture: The run time initializes S# models compiled by a slightly extended version of the C# compiler to ensure the desired S# semantics of required ports, port bindings, and fault effects. Both the simulator and the model checker use the run time to execute a model. The only difference between simulation and model checking is that the latter is exhaustive, checking all combinations of nondeterministic choices within a model whereas the former considers a single combination only. Model visualizations build upon the simulator, visualizing the simulated model state.

ment. The guidelines for systematic modeling introduced in Chapter 3 are supported by the S# modeling language, in particular for fault-related modeling activities. S# is a domain specific modeling language embedded into the ISO-standardized C# programming language, running on top of the ISO-specified .NET run time environment [105, 110]. S# is based on the model of computation introduced in Chapter 3: The execution of a .NET instruction corresponds to a micro step, so a macro step ends as soon as S# is finished executing all active behaviors that react to receptions of the Update signals.

Architecture Overview. The S# framework is based on two .NET libraries that can be used in any C#-based project: The core library contains the run time, the simulator, and the infrastructure for explicit-state model checking. Additionally, an extension library for the open source C# compiler Roslyn [149] checks for and reports S# modeling errors and does minor code transformations to implement S#'s semantics of faults and required ports. Model execution is unified, working in the same way regardless of whether a model is simulated or model checked. The only difference between non-exhaustive simulations and fully exhaustive model checking lies in the incomplete or complete enumeration of all combinations of nondeterministic choices within a model, respectively. S# provides no special visualization infrastructure to visualize a model other than the ability to use any framework available for C#, including, for instance, sophisticated GUI frameworks such as the Windows Presentation Foundation [147]. Figure 4.2 gives a brief overview of the individual parts that the S# framework consists of; more details of S#'s execution semantics are discussed in Chapter 6.

Models versus Programs. While S# models are represented as C# programs, they are still models of the safety-critical systems to be analyzed. However, S# somewhat weakens the classical distinction between programs and models which typically views models as abstractions over programs [198]. In the context of safety analysis, however, this distinction is in fact clear because the models have to describe the behavior of the controlled plants, whereas the plants are often not implemented as programs: In the height control case study, for instance, the vehicles are part of the model even though they are the plants the height control is designed to control, yet they are not designed, built, or developed in conjunction with the height control system. Additionally, some model parts represent hardware components, hydraulic or electrical subsystems, etc., none of which are software-based in the real system. Even those parts of the models that do in fact represent software components are generally not intended to be used as

C# Language Feature or .NET Functionality	Micro Steps	Macro Steps
controlled nondeterminism (`Choose` methods)	✓	✓
uncontrolled nondeterminism (threads, processes, …)	depends on use case	
managed heap allocations (`new` of reference types)	✓	×
unmanaged memory operations (via PInvoke)	✓	×
asynchronous methods (`async` and `await`)	✓	×
generator methods (`yield return`)	✓	×
lambda functions and anonymous delegates	✓	✓
dynamic typing (`dynamic`)	✓	×
static fields	×	×

Table 4.1. Overview of C# language features or .NET functionality that can be used during model simulations and model checking; all features not explicitly listed are fully supported. If a feature is only available within micro steps, it cannot be used in a way that spans multiple macro steps, that is, its computation must be complete before the end of the macro step that started it. For example, enumerating a generator method is supported, but the entire enumeration must take place within the same macro step. Similarly, an object newly allocated on the heap can be used by several micro steps, but no references to it can be persisted between multiple macro steps. Whether uncontrolled nondeterminism is acceptable depends on the use case.

the actual implementations of the real software: The models are usually an abstraction of the real software's behavior in order to increase analysis efficiency. Moreover, actual software implementations for safety-critical systems are typically written in the C or C++ programming languages [108, 111] for reasons of compatibility and performance; alternatively, synchronous languages such as Esterel might be used in order to get programming language support for the assumption of zero execution time [119, 189]. Off-nominal behavior is also only present in the S# models but is not implemented by the actual software components of a safety-critical system.

Usage Considerations. S# inherits C#'s language features and expressiveness and can use first- and third-party .NET libraries even when their original source code is not available. The general rule of thumb is that everything allowed by C# and .NET is also allowed by S#; for example, class inheritance, interfaces, virtual dispatch, generics, lambda functions, among most other features, are fully supported. Table 4.1 shows some of the advanced C# and .NET framework features that can only be used within micro steps but are not allowed to span multiple macro steps. Most C# features can indeed be used during micro step execution, showing the high level of expressiveness supported by S# models. The biggest exceptions are arbitrary memory allocations and uncontrolled nondeterminism as discussed in the following.

Memory Allocations. S# is unable to store newly allocated objects within a model's state during model execution, that is, any objects that are intended to be used during model execution must be preallocated. Transient memory allocations, on the other hand, are indeed supported, i.e., new objects that are allocated during a macro step are unproblematic as long as they are not persisted across multiple macro steps and are therefore cleaned up automatically by the .NET garbage collector. Transient allocations

are often encountered when using advanced C# features or .NET libraries such as lambda functions or LINQ methods that implicitly cause heap allocations. This restriction is rooted in a technical decision in the implementation of S# that might be revised in the future; it is certainly possible to add support for arbitrary object creation and storage to S# at the expense of a more complicated, less efficient implementation.

Uncontrolled Nondeterminism. Micro steps must be deterministic during model checking, otherwise S# is unable to guarantee that all possible behaviors are enumerated or it might fail to create counterexamples. The only source of fully supported nondeterminism are invocations of S#'s own family of `Choose` methods that are the primary mechanism for nondeterministic S# models. Sources of uncontrolled nondeterminism during model execution that lie outside of S#'s reach are usages of threads and locks, process spawning, random number generation, and so on. The possibility to invoke uncontrolled nondeterministic behavior is nevertheless important for some models, e.g., the self-organizing robot cell case study integrates an actual constraint solver into the model that is invoked during model execution to compute new system configurations, cf. Chapter 8. While such process spawning is a highly nondeterministic procedure that can fail for various reasons, a well-tested implementation yields mostly deterministic results and can thus be acceptable for use during model execution; in case of failure, an exception can be thrown that S# is able to handle and report back during model checking. Consequently, uncontrolled nondeterminism can in fact be useful as long as the result is either highly likely to be deterministic for some reason or else an exception is thrown; otherwise, model checking silently becomes non-exhaustive. For model simulations, on the other hand, uncontrolled nondeterministic behavior is likely undesired if a simulation is used as a regression test, but it might be acceptable in interactive visualizations. In any case, the S# framework does not attempt to restrict models in any way, on the one hand because of its design philosophy and on the other hand because it cannot do so: Guaranteeing determinism would require overapproximating analyses of compiled .NET code that might render valid use cases impossible.

4.2 Structural Modeling

Like SysML, S# supports both component-oriented and object-oriented modeling concepts as well as data types such as lists or hash tables. C#, on the other hand, is first and foremost an object-oriented language, so S#'s component-related concepts have to be mapped onto C#'s object-oriented features. Doing so requires the help of the types in the `SafetySharp.Modeling` namespace declared by S#'s core library that constitute the domain specific language embedded into C# [198]. Concepts that cannot be natively mapped to standard C# constructs in a convenient way, such as connections between required and provided ports, make use of some minor code transformations automatically carried out by S#'s extensions to the C# compiler. All in all, S#'s modeling concepts are closely related to those of SysML and are embedded into C# in such a way that they do not feel alien to experienced C# developers. Thanks to the automatic code transformations of the S# compiler, syntactic overhead within the models is relatively minor and is generally made up for by S#'s sophisticated tooling inherited from C#, including code completion and navigation, refactorings, as well as debugging capabilities.

In S#, the structure of a system is modeled as a non-strict hierarchy of components with explicit communication ports. S# components are represented by C# classes derived from the S#-provided type `SafetySharp.Modeling.Component`. Component instances are consequently represented by objects, i.e., by instances of the corresponding `Component`-derived classes. The PressureSensor component of the pressure tank case study from Figure 3.9 on page 35, for instance, is declared in S# as follows:

```
using SafetySharp.Modeling;
class PressureSensor : Component
{
   // ...
}
```

The declaration of the `PressureSensor` C# class represents a S# component type, whereas an instantiation of the class, i.e., `new PressureSensor()`, creates a S# component instance of the `PressureSensor` component type. In the remainder of this thesis, the namespace `SafetySharp.Modeling` is always assumed to be imported, that is, the `using` directive in the first line above remains implicit in all subsequent S# listings.

4.2.1 Component State and Subcomponents

The state information of a component consists of all fields that the component implicitly or explicitly declares. Fields can be of any .NET type, only classes that rely upon native operating system resources are generally problematic as they introduce uncontrolled nondeterminism, e.g., `Thread`, `Socket`, `File`, and so on. The following .NET types are consequently supported in S# models:

- primitive types such as `bool`, `int`, `uint`, pointers, object references, ...,
- enumeration types,
- struct types consisting of fields of supported types,
- delegate types,
- interface types,
- arrays of supported types, and
- class types consisting of fields of supported types such as `List<T>`.

As components usually hide their state from other components, fields are typically declared with `private` accessibility. The following contrived example shows a component declaring a field `_c` that stores a reference to some other `Component` instance, an integer field `_i`, a Boolean array field `_b`, and an additional, implicitly declared `private` field of type `bool` for the auto-implemented property `P` [105]; as explained later, `P` additionally declares two `public` provided ports. Except for field `_c`, all of the fields can store different values while the model is executed; `_c`, by contrast, is declared as `readonly`, meaning that its value, i.e., the reference to the other `Component` instance, can only be set in `C`'s constructor as guaranteed by the C# compiler.

```
class C : Component
{
   [Subcomponent] private readonly Component _c;
   private int _i;
   private bool[] _b;
   public bool P { get; set; }
}
```

In SysML, regular associations and composite associations between blocks are used to distinguish between blocks that simply reference each other and blocks that are a physical part of another block [153]. To express this difference in S#, the `[Subcomponent]` attribute can be used. Consequently, fields of a `Component`-derived type such as `_c` above are considered to declare a subcomponent relationship when they are marked with the `[Subcomponent]` attribute provided by the S# framework. Without this attribute, an instance of `C` would simply reference the component instance stored in `_c` without any subcomponent relationship existing between them.

It is occasionally useful to declare certain states as illegal, causing them to be ignored during model checking. In the height control case study, for instance, it is physically impossible for more than one vehicle to share the same physical location. Yet this constraint is surprisingly hard to encode in a control flow- or state machine-based way, regardless of whether the description is attempted locally for each vehicle or globally for the set of all vehicles. On the other hand, it is significantly easier to describe such constraints as Boolean expressions. Therefore, S# allows the specification of such state constraints as expressions in its models, letting the model checker take care of ignoring such states. A contrived example for a state constraint forbidding field `_x` from ever having the value 17 is shown below; more complex constraints are also possible, as the `AddStateConstraint` method also accepts arbitrary delegates, i.e., arbitrary C# code can be executed to evaluate state constraints. Chapter 7 shows the state constraint for the height control case study.

```
class C : Component
{
   private int _x;

   public C()
   {
      AddStateConstraint(_x != 17);
   }
}
```

4.2.2 Range Restrictions

Range restrictions are a declarative way to restrict the allowed range of the values that can be stored in a field. For example, an integer field `_f` might only be allowed to contain values that lie within the range of 0 to 5, inclusively. With range restrictions, the S# framework can automatically ensure that only the values 0, 1, 2, 3, 4, or 5 can be stored in `_f` at the end of each macro step; different micro steps during the same macro step can see values in `_f` that violate the range temporarily. How such violations are resolved is determined by the `OverflowBehavior`: If `Clamp` is specified, values falling outside of the range are clamped to the upper or lower bound of the range, whichever is closer. `WrapClamp`, by contrast, does the same as `Clamp` but uses the opposite bound instead. `Error` indicates that S# should throw a `RangeViolationException` whenever the field's range is violated. Such exceptions can be checked for using model checking, which also generates a witness that shows how such a range violation can be reached.

S# provides two mechanisms to declare ranges and overflow behaviors: If the range boundaries are compile time constants, the `Range` attribute can be used to declaratively

specify the range restriction. Sometimes, however, it is not possible to statically determine the range of a field, so alternatively it is possible to specify range restrictions dynamically in a component's constructor. The following example shows both ways to limit the ranges of the fields `_f1` and `_f2`. The upper bound of `_f2`'s allowed range is determined dynamically when `C` is instantiated; it is passed as a parameter to `C`'s constructor, hence the attribute-based approach that works for `_f1` cannot be used to restrict the range of `_f2`.

```
class C : Component
{
   [Range(0, 5, OverflowBehavior.Clamp)]
   int _f1;

   int _f2;

   public C(int upperBound)
   {
      Range.Restrict(_f2, 0, upperBound, OverflowBehavior.Error);
   }
}
```

4.2.3 Port Declarations

All methods and properties of a component are considered to be either required or provided ports depending on whether they are declared as `extern` and have a body; properties implicitly define methods that correspond to getter and setter functions in other programming languages [105]. C#'s regular method invocation syntax or property reads and writes represent invocations of required and provided ports. A component declares provided ports as regular C# methods or properties; they can optionally be `virtual` or `abstract` to support component inheritance. Required ports are also declared as regular C# methods or properties, albeit using the `extern` keyword without a body; they too can optionally be `virtual` to support component inheritance, while `abstract` required ports are nonsensical as they inherently declare no body. The following contrived example shows several different ways of declaring provided and required ports, using a variety of C# features that S# also supports.

```
class C : Component
{
   public virtual bool P1 => /* ... */;
   public bool P2
   {
      get { /* ... */ }
   }

   public int P3 { get; set; }

   public int P4(out bool x)
   {
      // ...
   }

   public extern virtual bool R1 { get; }
   public extern bool R2 { get; set; }
   public extern int R3(ref bool x);
}
```

The first two provided ports `bool get_P1()` and `bool get_P2()` are declared as two getter-only properties `P1` and `P2` of type `bool` with an expression body and a statement body, respectively [105]. The auto-implemented property `P3` implicitly declares an `int` state field and two provided ports, namely the getter `int get_P3()` and the setter `void set_P3(int value)`. The last provided port `P4` is an `int`-returning method with a Boolean `out` parameter `x`. The first required port `bool get_R1()` is declared as a getter-only property `R1` of type `bool`. The property `R2` effectively declares the required ports `bool get_P2()` and `void set_P2(bool value)` without introducing a corresponding state field. The last required port `R3` is an `int`-returning method with a Boolean `ref` parameter. `P1` and `R1` are declared as `virtual`, allowing component types derived from `C` to override their implementation or connection to a provided port, respectively. All of the ports in the example have `public` accessibility, allowing other component types to invoke or connect to them; accessibility of ports is no different than regular C# accessibility for methods and properties [105].

4.2.4 Port Bindings

Port bindings in S# correspond to port connections in SysML. As required ports have no body, they cannot be invoked unless they are bound to a provided port with a compatible signature. Required ports therefore forward all invocations to the provided port they are bound to; invoking an unbound required port is an error in which case the S# framework raises an `UnboundPortException` during model execution. In S#, port bindings are established by calling the `Bind` method, using C#'s `nameof` operator to pass the names of the required and provided ports that should be bound. Ports are bound by name to work around several C# language limitations; the S# compiler ensures that the bindings are established between ports of compatible signature and rewrites the invocation of the `Bind` method such that the actual `Component` instances that are bound are also passed along. Ports are always bound for concrete component instances, never for component types. That is, the `Bind` method in the following example binds the required port `R` of the component instance `a1` to the provided port `P` declared by component instance `a2`, where `a1` and `a2` are passed to `C`'s constructor. Once bound, an invocation of `a1.R` indirectly invokes `a2.P` such that `true` is returned by `a1.R`.

```
class A : Component
{
   public bool P() => true;
   public extern bool R();
}

class C : Component
{
   public C(A a1, A a2)
   {
      Bind(requiredPort: nameof(a1.R), providedPort: nameof(a2.P));
   }
}
```

Similar to how direct invocations of provided operations are used in the internal block diagram shown in Figure 3.10 on page 37, S# allows provided ports of other component instances to be invoked directly. In the example above, component `C` is allowed to invoke

`a2.P` directly, that is, it does not have to call a required port bound to `a2.P`. While such shortcuts potentially break encapsulation and compositionality, they help to reduce clutter in the models by making them more concise. Whether these shortcuts are used is a matter of style and eventually comes down to personal preference. In this thesis, direct calls are used when parent components access subcomponents, while port bindings are preferred between sibling components, and, in particular, between plants and sensors or actuators.

4.3 Behavioral Modeling

In S#, behavior is represented by C# code that is executed during model simulations and model checking. S# supports two kinds of behavior: Control flow-based behavior in the form of regular C# statements and expressions as well as state machine-based behavior that builds upon the `StateMachine<T>` type provided by the S# core library. The global behavior of a model is implicitly defined by the local behaviors of the individual components, their ports, as well as their communication interrelationships via port bindings. The individual instructions that the C# code is compiled into represent micro steps, therefore a macro step typically consists of quite a few C# statements and expressions. The following example is a partial model of the pressure sensor of the pressure tank case study that omits any faults and fault effects:

```
class PressureSensor : Component
{
   public virtual bool IsFull => PhysicalPressure >= Model.SensorFullPressure;
   public virtual bool IsEmpty => PhysicalPressure <= Model.SensorEmptyPressure;

   public extern int PhysicalPressure { get; }
}
```

The provided ports `IsFull` and `IsEmpty` check whether the tank's pressure exceeds the sensor's reporting limits. The pressure is checked against the value retrieved from the `PhysicalPressure` required port that is bound to the tank component. The reporting limits are given by the global constants `Model.SensorFullPressure` and `Model.SensorEmptyPressure`. Alternatively, the limits could have been specified as local constants within `PressureSensor` itself or they could have been passed to the sensor as constructor arguments. All three alternatives yield the same analysis results, they only differ in terms of model comprehensibility and composability. Collecting all global constants in the `Model` class allows them to be viewed and changed in unison but is only possible because of the case study's simplicity that does not require any complex model composition.

S# supports nondeterministic models, checking all possible combinations of nondeterministic choices during model checking. However, only nondeterminism that is actually known to S# can be exhaustively checked as discussed previously in Section 4.1. Therefore, the only source of nondeterminism within a model should be uses of S#'s own family of `Choose` methods; as explained in Chapter 6, these methods allow the S# run time to exhaustively enumerate all possible combinations of nondeterministic choices. The following simple example shows how the `Choose` method can be used to nondeterministically return either `true` or `false` from a provided port. Based on the `Choose` method, S# provides some additional helper methods that are

occasionally useful: The `ChooseFromRange(lowerBound, upperBound)` method, for example, nondeterministically returns a value that lies within the given input range, while `ChooseIndex(elementCount)` can be used when working with lists or arrays to nondeterministically select a valid index.

```
class C : Component
{
    public bool P => Choose(true, false);
}
```

4.3.1 Active Behavior

Components can optionally have active behavior that is triggered automatically by Update signals during the execution of a macro step as discussed in Chapter 3. In S#, receptions of the Update signal are represented by invocations of the `Update` method declared by the `Component` base class that all components are allowed to override. During model execution, the S# framework calls the `Update` methods of all top-level components the model consists of which are subsequently free to distribute the signal among their subcomponents. If a component does not override the `Update` method, it conceptually has no active behavior and therefore does not react to receptions of the Update signal. Nevertheless, it is best practice to always call the `Update` method of all components, regardless of whether they actually have an active behavior; this way, overriding the `Update` method becomes an implementation detail of each component and future additions of active behavior will work correctly. For example, the S# version of the pressure controller of the pressure tank case study can be declared as follows:

```
class PressureController : Component
{
   [Subcomponent] public readonly Pump Pump;
   [Subcomponent] public readonly Timer Timer;
   [Subcomponent] public readonly PressureSensor Sensor;
   [Subcomponent] public readonly SoftwareController Controller;

   // constructor omitted

   public override void Update()
   {
      Update(Sensor, Pump, Timer, Controller);
   }
}
```

Only the `Timer` and `SoftwareController` components actually override the `Update` method, whereas the `Pump` and `PressureSensor` components do not. Nevertheless, the `PressureController` component invokes the `Update` methods of all four subcomponents, as otherwise the controller would be tightly coupled to the concrete types of the `Pump` and `PressureSensor` instances that do not have any active behavior. Therefore, the controller could not be reused, for instance, when a different sensor type derived from `PressureSensor` is available that actually overrides `Update`, even though all provided and required ports would otherwise be the same. The order in which Update signals are distributed through the model might also be relevant; in the example above, the timer is updated before the software controller gets a chance to do the same, though in this case the actual ordering is irrelevant for formal safety analysis.

The active behavior of the Tank component printed below shows the discretization of the differential equation declared by the PressureEquation constraint in Figure 3.18 on page 46, updating the pressure level depending on the amount of incoming and outgoing fluid during each macro step. The actual pressure level within the tank is stored in the _pressureLevel field whose value is simply returned by the PressureLevel provided port. The provided ports IsRuptured and IsDepleted are used to specify the hazards but are otherwise not invoked by any other components. The IsBeingFilled required port allows the tank to determine whether the pump is enabled; if so, the pressure level increases, otherwise it decreases in accordance with the range restriction specified via the attribute on the _pressureLevel field. Once the tank has ruptured, its active behavior is effectively disabled as it no longer has any side effects. The _pressureOut field is a constant as the amount of outgoing fluid is not considered in detail whereas the value of the _pressureIn field depends on whether the pump is enabled.

```
class Tank : Component
{
   [Range(0, Model.PressureLimit, OverflowBehavior.Clamp)]
   private int _pressureLevel;

   private int _pressureIn = 0;
   private const int _pressureOut = -1;

   public bool IsRuptured => _pressureLevel >= Model.PressureLimit;
   public bool IsDepleted => _pressureLevel <= 0;

   public int PressureLevel => _pressureLevel;

   public extern bool IsBeingFilled { get; }

   public override void Update()
   {
      if (!IsRuptured)
      {
         _pressureIn = IsBeingFilled ? 2 : 0;
         _pressureLevel += _pressureIn + _pressureOut;
      }
   }
}
```

4.3.2 State Machines

State machines are often encountered when modeling safety-critical systems, hence S# provides language support for a subset of the modeling features offered by SysML state machines. Compared to SysML, S# does not support history states, event queues, or state actions, for instance [153]. These limitations, however, are only of technical nature, so future versions of S# could be extended to support a larger subset of SysML's state machine modeling features, if necessary. On the other hand, S# integrates state machines into the overall control flow of the models, as the S# modeling language is primarily control flow-based due to its C# heritage; for advanced scenarios, this integration offers some additional modeling flexibilities. The following example is the S# version of the SysML state machine shown in Figure 3.17 on page 46, that is, the active behavior of the pressure tank case study's software controller:

```
class SoftwareController : Component
{
   // constructor and declarations of the _pump, _timer, and _sensor fields omitted

   private enum State
   {
      Inactive,
      Filling,
      StoppedBySensor,
      StoppedByTimer
   }

   private readonly StateMachine<State> _stateMachine = State.Inactive;

   public override void Update()
   {
      _stateMachine
         .Transition(
            from: State.Filling,
            to: State.StoppedByTimer,
            guard: _timer.HasElapsed,
            action: () => _pump.Disable())
         .Transition(
            from: State.Filling,
            to: State.StoppedBySensor,
            guard: _sensor.IsFull,
            action: () =>
            {
               _pump.Disable();
               _timer.Stop();
            })
         .Transition(
            from: new[]
               { State.StoppedByTimer, State.StoppedBySensor, State.Inactive },
            to: State.Filling,
            guard: Sensor.IsEmpty,
            action: () =>
            {
               _pump.Enable();
               _timer.Start();
            });
   }
}
```

The individual states of a state machine are typically enumeration literals or, in more advanced scenarios, any other type. The `State` enumeration declared above names the four states of the state machine implementing the `SoftwareController`'s active behavior. The `_stateMachine` field of type `StateMachine<State>` is declared as `readonly`, thus the reference to the state machine can never change during model execution. The state machine in turn has a field that stores its current state, hence the `SoftwareController`'s state conceptually consists of the state machine's state field. The `Inactive` state is the state machine's initial state as indicated by the conversion assignment in the field declaration above; it is also possible to assign multiple initial states using C#'s array syntax, one of which is then chosen nondeterministically.

The `Update` method of the `SoftwareController` represents the controller's active behavior. As all transitions of the state machine shown in Figure 3.17, page 46, react

to receptions of the Update signal, all five transitions are declared within `Update` by invoking the `Transition` method on the `_stateMachine` field. A transition consists of one or more mandatory source states, one or more mandatory target states, and optional guards and actions specified in the `from`, `to`, `guard`, and `action` parameters, respectively. If the guard is omitted, it is always assumed to be `true`; if the action is omitted, it is assumed to be `() => {}`, i.e., the empty action without any effects. Guards are specified as Boolean expressions whereas actions are described by arbitrary lambda functions with side effects, nondeterministic choices, port invocations, and so on.

A transition is only enabled when the state machine is currently in one of the transition's source states and its guard holds. If multiple transitions are enabled, one is chosen nondeterministically. If a transition has multiple target states and the transition is enabled and chosen, one target state is chosen nondeterministically. The chosen transition's optional action is executed after the state has been updated. The event that triggers a transition is not directly represented in the `Transition` method; instead, it is provided by the context the transition is declared in. In the example above, all transitions are declared in the component's `Update` method, hence all transitions are conceptually considered to be triggered by receptions of the Update signal. It is also possible to declare some or all transitions in provided ports, meaning that the transitions can only be taken when that port is called, the state machine is in one of the transitions' source states, and the transitions' guards hold. Such transitions are also able to reference parameters of the provided port in their guards or actions. The following contrived example illustrates a state machine that leaves the state `S0` and enters state `S1` whenever the provided port `P` is invoked with a value greater than zero passed to its parameter `x`:

```
public void P(int x)
   => _stateMachine.Transition(from: State.S0, to: State.S1, guard: x > 0);
```

4.4 Fault Modeling

In S#, faults and their effects are modeled analogously to their specification in SysML, cf. Figure 3.24 on page 55. Conservative extension, i.e., the preservation of the nominal behavior before fault injection, is guaranteed as fault effects are only enabled when their associated faults are activated. Error propagation remains implicit in S# models to simplify modeling as are failures that cause input faults in other components through port bindings, for instance; these propagation chains are what S# is designed to uncover automatically. To add a fault to a model, a field of a `Fault`-derived type has to be added to a component. If the fault affects a single component only, it is usually advisable to add the fault to the affected component itself, as demonstrated for the `PressureSensor` component in the following example. As shown below, faults can be initialized with any object of a `Fault`-derived type with the two most common kinds of persistency provided by S#'s core library, namely `TransientFault` and `PermanentFault` for transient and permanent persistency, respectively. It is also possible to implement custom `Fault`-derived types to model other kinds of fault persistency as explained in the S# Wiki [102] using the example of permanent persistency with maintenance. Fault fields are usually both `public` and `readonly` so that faults can be referred to in model visualizations, for instance, but they cannot be reinitialized from outside the component.

```
class PressureSensor : Component
{
   public readonly Fault SuppressIsEmpty = new TransientFault();
   public readonly Fault SuppressIsFull = new TransientFault();

   public virtual bool IsFull => PhysicalPressure >= Model.SensorFullPressure;
   public virtual bool IsEmpty => PhysicalPressure <= Model.SensorEmptyPressure;

   public extern int PhysicalPressure { get; }

   [FaultEffect(Fault = nameof(SuppressIsFull))]
   public class SuppressIsFullEffect : PressureSensor
   {
       public override bool IsFull => false;
   }

   [FaultEffect(Fault = nameof(SuppressIsEmpty))]
   public class SuppressIsEmptyEffect : PressureSensor
   {
       public override bool IsEmpty => false;
   }
}
```

Fault effects are commonly modeled by adding nested classes derived from the affected component such as `SuppressIsFullEffect` and `SuppressIsEmptyEffect` above. A fault effect must always be marked with the `[FaultEffect]` attribute that typically links the effect to its associated fault by statically setting the attribute's `Fault` property to the fault's name. Fault effects can override provided ports, required ports, or the `Update` method to describe the local effects of the fault on the affected component. In the example above, the `IsFull` and `IsEmpty` provided ports are declared as `virtual` such that they can be overridden by the two fault effects using C#'s `override` modifier. Consequently, fault effect modeling builds upon C#'s support for class polymorphism and .NET's standard virtual dispatch mechanism. On top of that, the S# compiler ensures the correct execution semantics through minor code transformations so that an effect is only executed when its associated fault is activated.

For example, when the `IsFull` port is invoked, either directly or indirectly via a port binding, the original port behavior declared by the `PressureSensor` component is only executed if `SuppressIsFull` is not activated. Conversely, when the S# framework activates the fault during model execution, the off-nominal behavior declared by `SuppressIsFullEffect` is executed instead of the original nominal behavior, in this case always returning `false` regardless of the actual physical pressure level in the tank. Consequently, the fault's effect causes a failure of the sensor in that it no longer reports the pressure tank to be full. The fault effect for the `SuppressIsEmpty` fault works in a similar way, affecting the `IsEmpty` port instead. The above example only shows how fault activations immediately result in observable component failures without first introducing non-observable errors. It is of course also possible, especially when overriding the `Update` method, that fault activations result in errors that are not immediate failures: The off-nominal active behavior could change some values of the component's fields, for instance, resulting in latent errors. The S# framework subsequently determines automatically how such an error propagates through the component and whether it eventually becomes an externally observable failure or potentially disappears entirely.

There can be multiple fault effects affecting the same component port. In that case, however, the S# compiler by default issues a warning because the fault effect that actually has an effect is chosen nondeterministically, which is often unintended. The nondeterminism can be statically resolved by marking both fault effects with the `[Priority]` attribute, giving priority to one of the effects. That is, if both fault effects are enabled, only the effect with the higher priority is considered. If the nondeterminism is indeed intentional, both fault effects can be explicitly marked with the same priority level, e.g., `[Priority(0)]`, to silence the warning and to allow S# to choose between the two effects nondeterministically.

A fault can affect multiple component instances in order to model common cause faults, for instance, or in order to speed up safety analyses as shown for the height control case study in Chapter 7. In that case, it is typically advisable to add the fault to a parent component of all affected components and pass the reference to the fault to the affected components' constructors as shown below. There is a $1 : n$ correspondence between faults and fault effects, that is, a fault can have multiple effects, but each effect belongs to exactly one fault. There are two ways to associate faults and fault effects: Either statically at compile time as shown above or dynamically at run time as shown below. Statically, a fault effect is associated with a fault by name matching via the `[FaultEffect]` attribute; it is the preferred approach for faults that affect a single component only. If, by contrast, the association between a fault and its effects cannot be statically determined at compile time, the `Fault.AddEffect<TEffect>(Component)` method can be called dynamically when the model is initialized. In the following example, this method is used to associate the effect of type `E` given as a generic parameter with the `fault` passed to `C`'s constructor, affecting the instance of `C` the constructor is executed for:

```
class C : Component
{
   public C(Fault fault)
   {
      fault.AddEffect<E>(this);
   }

   [FaultEffect]
   public class E : C
   {
      // ...
   }

   // ...
}
```

4.5 Model Composition

A S# model composes together individual component instances, denoting a set of top-level or root components that represent the modeled system's controllers and plants. Models can be composed together using arbitrary C# code; how a model is constructed is of no interest during model execution and analysis. For example, valid design variants could be stored in a database where they are retrieved from and assembled dynamically. It is also possible to create a custom textual or graphical language for the definition of

model configurations that is integrated into S# through a parser. The design variants of the height control case study, on the other hand, are composed together using reflection as shown in Chapter 7. Components can be parameterized in three main ways to allow for flexible model composition: They can have generic type parameters, for instance allowing for substitution of subcomponent or field types. Constructor parameters can influence ranges, the setup of faults and fault effects, as well as the creation of subcomponents and port bindings, among many other things. Additionally, component inheritance or component interfaces can be used to swap out parts of a model with different structural or behavioral design variants.

A model is declared as a class derived from S#'s `ModelBase` class as shown below. Models are assumed to declare properties marked with the `[Root]` attribute that declare the model's top-level components, but are otherwise free to contain arbitrary C# code to flexibly compose together different variants of a model. For the pressure tank case study model shown below, however, only the default design is considered, so there is no complicated model initialization logic. The global constants declared by the model are used by its constituting components in order to parameterize certain aspects of the overall model. They are declared in the `Model` class so that they can be viewed and changed in unison, while more complicated models with various design variants typically do not use global constants like this: When different variants of a model make use of different configuration values, these values must often be passed via constructors to the respective components, for example.

```
class Model : ModelBase
{
   public const int PressureLimit = 60;
   public const int SensorFullPressure = 55;
   public const int SensorEmptyPressure = 0;
   public const int Timeout = 59;

   [Root(RootKind.Plant)]
   public Tank Tank { get; } = new Tank();

   [Root(RootKind.Controller)]
   public PressureController Controller { get; } = new PressureController();

   public Model()
   {
      Bind(nameof(Controller.Sensor.PhysicalPressure), nameof(Tank.PressureLevel));
      Bind(nameof(Tank.IsBeingFilled), nameof(Controller.Pump.IsEnabled));
   }
}
```

The two getter-only properties store the model's `Tank` and `PressureController` instances; as is idiomatic in both C# and S#, properties are often named the same as their underlying types. The declared properties are inline-initialized with new `Tank` and `PressureController` instances; every read of the properties therefore returns the instances automatically created during the construction of the `Model` instance [105]. The `Tank` property is marked with the `[Root(RootKind.Plant)]` attribute that indicates to the S# framework that the `Component` instance stored in the property is part of the analyzed model's set of plants. Conversely, the `Controller` property is marked with the `[Root(RootKind.Controller)]` attribute to indicate that the stored `Component` instance

is part of the analyzed model's set of controllers. It is important for the S# framework to know the distinction between top-level plants and controllers as the `Update` methods of all plants must be invoked before the `Update` methods of the controllers. Whether a `Component` instance is considered a plant or a controller is model-specific, as different views of the analyzed system might require these role allocations to change. Additionally, the model's constructor establishes the port bindings that correspond to the connections between the different ports shown in the internal block diagram of the pressure tank case study in Figure 3.10, page 37.

4.6 Related Work

Embedded versus Standalone Domain Specific Languages. Building a modeling and analysis framework for safety-critical systems on top of a standard programming language is somewhat unorthodox. One of the few other modeling languages taking such an approach is SystemC [73] that is based on the ISO-standardized C++ programming language [111]; in contrast to S#, however, SystemC was initially not designed for formal analyses, though such extensions exist [39]. The main benefits of embedding S# into C# lie in the inherited tooling and infrastructure support with sophisticated code completion and code navigation capabilities, various code refactorings that are also applicable to S# models, debugging facilities, and continuous integration support, among others. In particular, S# can build upon the open source compiler Roslyn [149] to access abstract syntax trees and to conduct semantic analyses of the S# models with extensive error recovery mechanisms that reduce implementation effort and increase S#'s overall usability. Most notably, basic language features such as the syntax and semantics of expressions and statements are already specified and implemented, allowing S# to focus on those aspects of its models that actually matter for safety analysis, namely fault modeling. Custom languages such as the ones used by VECS or the HATS project [40, 135], by contrast, require large amounts of work just to implement basic functionality like expression parsing and type checks. Additionally, S# benefits from future extensions of C#: The next version is likely to support local functions, tuples, and a simple form of pattern matching, with more advanced pattern matching capabilities scheduled for later releases [176]. While these features might require small changes to S#'s compiler extensions, they are trivially supported during model execution.

S# Semantics & Model Transformations. The S# framework does not rely upon complex model transformations, only the compiler performs some minor code transformations that shift some language constructs around without fundamentally changing the model's level of abstraction or the expressiveness of the modeling language. Hence the implementation of these transformations is relatively straightforward, in particular compared to the transformations carried out by the Compass toolset [151] in order to transform its high-level AADL models to the low-level NuSMV input language [38]; the transformations' complexity results from the completely different levels of expressiveness between these two languages. On the other hand, S# models are also transformed, namely by the C# compiler into intermediate language [110] and then by .NET's just-in-time compiler to native CPU instructions. However, both transformations are used by millions of programs world-wide and therefore have extremely high test coverage,

making bugs unlikely even though not completely impossible. It is always necessary to have faith in the correctness of the compilers as well as the CPUs that execute both the compilers as well as the compiled programs, a problem that international norms for the development of safety-critical systems [107] try to work around by requiring the use of multiple tool chains and hardware components in very safety-critical scenarios. From a formal methods point of view, however, the biggest threat to S#'s semantic consistency is the lack of a formal specification: The official C# specification [105] is only available in natural language; older versions of C#, however, have been formalized by Börger et al. [22]. In contrast to C and C++ [108, 111], C# does not have any intentionally undefined behavior in its specification [105]. Nevertheless, it does not reach the same level of formalization as the VECS or Compass toolsets [135, 151], for instance.

Model Composition. Support for modeling design alternatives differs widely between the different languages used for formal safety analysis. The S# framework is one of the tools that has the most comprehensive support in this area, allowing for component inheritance and interfaces, generic components, and component parameterization via constructor arguments. VECS [135], by contrast, only allows for parameterization of constants and replacement of components, though components are not fully encapsulated and therefore complicate component exchange [188]. The AADL language used by the Compass toolset [151], on the other hand, requires component interfaces to be specified and supports full component encapsulation, though the publicly accessible case studies do not make use of these functionalities; the large but confidential satellite case study conducted by the European Space Agency [58] likely takes advantage of AADL's modeling capabilities in this area. FSAP/NuSMV and its successor xSap [24, 65] do not provide any real support for variant modeling. All in all, S# provides the most features and flexibility for modeling design variants compared to these other tools. Only SystemC [73] is comparable as it has all language features of C++ at its disposal; in particular, template meta-programming in C++ [3] is more powerful than S#'s support for generics, but in contrast to SystemC, S# can make use of reflection [110] as shown in Chapter 7. Moreover, S# and SystemC can also automate model composition in contrast to all other aforementioned tools: Arbitrary C# code can be executed to instantiate components and to connect them together in order to build up the overall model, thereby providing meta-constructs for model creation. As discussed in Chapter 8, these capabilities are particularly useful for virtual commissioning and safety analyses of self-organizing systems.

Fault Injection via Delta Modeling. The modeling languages of the HATS project [40] are custom languages with both functional and object-oriented features; they are integrated into the Eclipse development environment. The languages are executable through transformations to Java [72] and formally analyzable through transformations to the Maude system [141] that is based on rewriting logic. HATS is not primarily concerned with safety analysis but rather with modeling spatial and temporal variability. Its support for variability modeling is based on the delta modeling approach and requires various different modeling languages to describe all variants of a system [85]. Delta modeling or delta-oriented programming was originally introduced by Schaefer and Damiani [177] as a novel approach for modeling product lines [16]. It is a modular

and flexible technique for adding, removing, or modifying parts of a model. In the context of formal safety analysis, the delta modeling concepts could alternatively be used to model fault effects on system components by adding, removing, or changing system behavior as opposed to the approach taken by S# based on virtual dispatch. A feature very similar to delta modeling is planned for a future version of C# [176], also enabling method replacement, albeit for completely different reasons: Code generation by UI frameworks, for instance, often creates additional parts of a class, sometimes requiring the possibility to replace a handwritten method by a generated one or vice versa; the replacing method is free to invoke the replaced one if necessary. Once this method replacement mechanism is indeed available in a future version of C#, it could be used as an alternative way to model fault effects, using the new mechanism instead of the current virtual dispatch-based approach. However, both the planned method replacement feature as well as the delta modeling technique are static in the sense that the modifications made by them to a model are permanent, hence extensions would be necessary to support the dynamic nature of fault activations. Such extensions are likely to be conceptually similar to the way the S# compiler transforms the current virtual dispatch-based fault injection approach.

Summary and Outlook. The S# modeling and analysis framework for safety-critical systems is based on the C# programming language and the .NET run time environment in order to allow systematic modeling in accordance with the guidelines introduced in Chapter 3. As the faults that can affect the system to be analyzed are of primary interest during safety assessment, the S# modeling language provides first-class support for faults, different kinds of fault persistency, as well as the description of off-nominal behavior through fault effects. S# inherits the tooling from the .NET world, thereby combining the strengths of industrial software development environments and various tools for safety analysis; namely, the high level of expressiveness and tool support as well as rigorous formal analysis techniques, respectively. Most safety analysis tools developed in an academic context have a significantly lower level of usability compared to the software development tooling available to S# such as refactoring or debugging support. Analysis capabilities provided by the S# framework include model simulations, model checking, as well as model visualizations that are discussed in Chapter 6. The formal techniques that the S# framework makes use of are introduced separately in the next chapter as they are completely independent of S# and could thus be taken advantage of by other safety analysis tools. Chapter 7 shows the complete S# model of the height control case study and evaluates its safety, also modeling some of its design variants by taking advantage of S#'s advanced modeling features inherited from C# and the underlying .NET framework. In Chapter 8, the flexible model composition capabilities provided by the S# framework are used for run time analyses of self-organizing systems.

Summary. A new fault-aware modeling and specification approach for safety-critical systems amends Kripke structures and Linear Temporal Logic [13] with information about fault activations in order to increase explicit-state analysis efficiency by up to three orders of magnitude. Defined over these formal foundations, Deductive Cause Consequence Analysis (DCCA) is an established model checking-based safety analysis technique that computes all minimal critical fault sets for a hazard [80]. Conceptual improvements to DCCA enhance the model checking workflow as witnesses are generated to explain how critical fault sets can cause a hazard, which is more useful in practice than witnesses showing how non-critical fault sets cannot do so. Moreover, an additional variant of DCCA further reduces analysis times in many cases.

Publications. Fault-aware modeling and specification as well as the new DCCA variants are published in [83]; state-based fault modeling is outlined in [80].

5 Formal Safety Analysis

The focus of this chapter lies on basic and well-known modeling and specification formalisms, namely Kripke structures and Linear Temporal Logic (LTL) [13, 41, 133], that underlie established model checkers [13, 42] such as LTSmin, SPIN, or NuSMV [38, 98, 116]. As faults are obviously of particular interest in the context of safety analysis, Kripke structures and LTL are amended to conveniently express the concept of fault activations, thereby making faults and their effects explicit in the analyzed models. In contrast to the state-based view of faults that is commonly taken by various safety analysis tools [24, 135, 151], the extended formalisms consider fault activations [9], only taking situations into account in which faults actually have effects and cause errors. The activation-based view potentially reduces model checking times by an exponential factor compared to the state-based one; efficiency improvements of up to three orders

of magnitude can be observed for the height control case study, for instance. Section 5.1 introduces the notion of fault-aware Kripke structures, also reviewing the traditional state-based fault modeling approach. For formal fault modeling, two basic operations are formally defined and put into correlation in Section 5.2: Fault injection and fault removal. A definition of fault-aware LTL follows in Section 5.3 and Section 5.4 shows how fault-aware Kripke structures and LTL can be model checked.

Section 5.5 subsequently improves upon the original formalization of the model checking-based safety analysis technique Deductive Cause Consequence Analysis (DCCA) [80]. DCCA is specified over Kripke structures and LTL, making it a tool-independent formal safety analysis technique that is not tied to any high-level modeling formalisms or any analysis tool chains. DCCA rigorously assesses the safety of models of safety-critical systems by automatically computing all minimal critical fault sets for a hazard; these sets represent combinations of faults that can potentially cause the hazard. DCCA checks a series of LTL formulas with the help of a model checker [13, 42], thereby determining the cause consequence relationship between the faults (the causes) and the hazard (the consequence). By formalizing DCCA using the dual of critical fault sets, namely safe fault sets, a better model checking workflow results with a useful practical advantage compared to DCCA's original formalization: The model checker generates a witness showing how a critical fault set can actually cause the occurrence of a hazard instead of returning a witness that demonstrates how a safe fault set cannot do so. Witnesses for critical fault sets can subsequently be used to devise additional safety measures in order to improve system safety. Taking advantage of the fault awareness of the underlying Kripke structures, a variant of DCCA is defined that further reduces analysis times in many cases: Irrelevant faults can be removed when a fault set is checked for criticality, generating a smaller version of the analyzed fault-aware Kripke structure specifically tailored to the check.

Chapter 6 bridges the gap between executable models and the formal foundations presented in this chapter, discussing how executable models, including S# models, can take advantage of fault-aware modeling and specification as well as DCCA. Figure 5.1 gives an overview of the entire formal analysis approach spanning this and the subsequent chapter, showing the steps necessary to model check S# models and to assess their safety by carrying out DCCAs. Conceptually, executable models are first transformed to semantically equivalent fault-aware Kripke structures and fault-aware LTL formulas, which are subsequently converted to classical Kripke structures as well as classical LTL formulas. For these resulting Kripke structures and formulas, standard model checkers can be used for analysis. For reasons of efficiency, the S# framework employs a direct integration of the explicit-state LTSmin model checker [116] that allows for the execution of S# models during model checking. Consequently, no Kripke structures or LTL formulas ever have to be explicitly created during analyses of S# models. Nevertheless, the direct analysis approach used by the S# framework is rooted in the formal foundations established in the remainder of this chapter, in particular allowing for safety analyses of S# models with DCCA. Consequently, the S# framework is a tool that implements the formal modeling, specification, and safety analysis techniques introduced in the following, integrating the formal techniques into a high-level modeling approach.

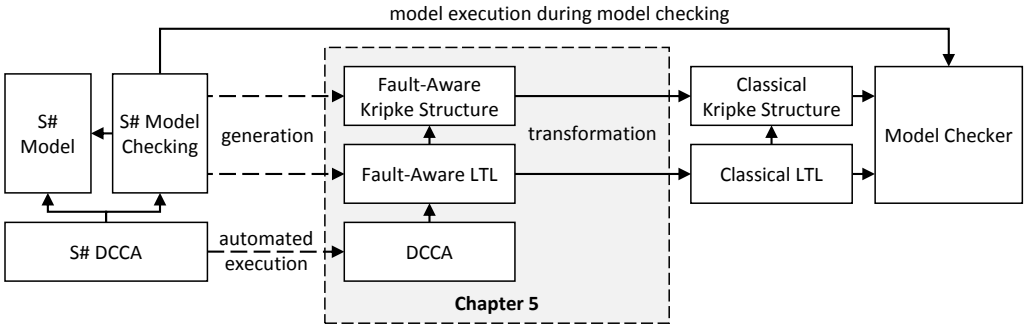


Figure 5.1. Overview of the relationship between Chapters 5 and 6: This chapter introduces fault-aware Kripke structures and fault-aware LTL over which DCCA is defined. Classical Kripke structures, classical LTL, and model checking are not discussed in greater detail in this thesis except for transformations of the fault-aware modeling and specification formalisms to their classical counterparts. Chapter 6 subsequently explains the integration of the S# framework into the formal foundations established in this chapter: Conceptually, fault-aware Kripke structures and fault-aware LTL formulas are generated from executable models, thereby given them Kripke structure semantics. For reasons of efficiency, however, the actual implementation of the S# framework employs a direct model execution approach instead of generating Kripke structures during explicit-state model checking with LTSmin [116] as discussed and evaluated in Chapter 6.

5.1 Formal Models of Safety-Critical Systems

Transition systems are a standard class of formal models that are used to describe the states and the behaviors of hardware and software systems [13, 42, 90]. They can be defined in many different ways ranging from low-level Kripke structures to complex high-level SysML state machines [121, 153, 154]. The latter expose more expressive modeling features that increase model comprehensibility at the expense of potentially making subsequent verifications less efficient due to the increase in semantic complexity. Kripke structures, on the other hand, are a well-known discrete-state, discrete-time modeling formalism [36, 41] that established model checkers such as LTSmin, SPIN, or NuSMV [38, 98, 116] are based on. Consequently, tools built for formal safety analyses such as S#, VECS, Compass, FSAP/NuSMV, xSAP, and others [24, 27, 65, 82, 122, 135] either implicitly or explicitly transform their models to Kripke structures for model checking, while their actual modeling formalisms are usually higher-level, i.e., more expressive. Formally [42], a Kripke structure $K = (P, S, R, L, I)$ consists of a set of atomic propositions P, a set of states S, a left-total transition relation $R \subseteq S \times S$, a labeling function $L\colon S \to 2^P$, and a non-empty set of initial states $I \subseteq S$.

For example, Figure 5.2 shows the Kripke structure that is generated from the nominal parts of the S# model of the pressure tank case study. The Kripke structure is completely deterministic, meaning that it has only one initial state and that all of its states have exactly one successor state. The Kripke structure therefore starts at its unique initial state where the tank is empty, which is eventually reached over and over once the tank has been fully filled up and completely depleted again. As Kripke structures have no built-in support for modeling time or duration, the concept of time remains explicit: The timer is modeled by labeling the states with numbers that indicate the amount of seconds

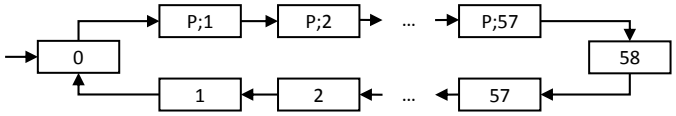


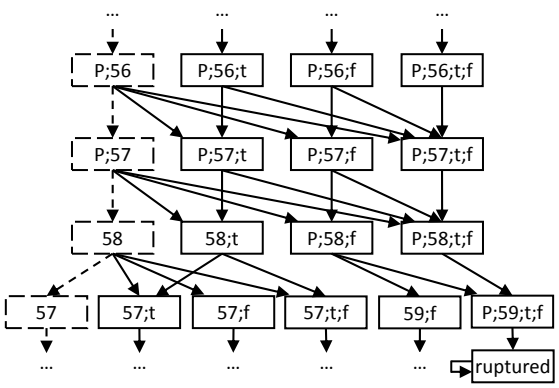
Figure 5.2. The Kripke structure modeling the pressure tank case study without any faults. Each box represents a state and contains the relevant atomic propositions that hold in the state: The label P indicates that the pump is running, whereas the number represents both the pressure level within the tank as well as the timer's value. The leftmost state is the initial one: The pressure in the tank is zero, and the sensor reports that the tank is empty. The next state shows that the pressure increases as the pump is running. Pumping continues until the rightmost state is reached, where pumping is stopped as the sensor reported the tank to be full. Subsequently, the pressure decreases until the tank becomes empty, in which case the initial state is reached again and the cycle starts anew.

that have passed since the timer's activation, so each transition implicitly represents the passing of one second. Due to the simplicity of the case study, its nominal behavior only consists of the cyclic sequence of tank filling and tank draining, in particular without ever reaching a state in which the tank is ruptured.

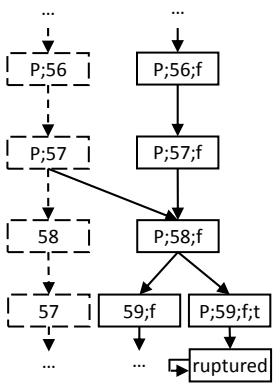
5.1.1 State-Based Fault Modeling

Figure 5.3a shows a simplified part of the pressure tank case study's Kripke structure with off-nominal behavior shortly before the tank is fully filled and either the pump is shut off or the tank ruptures. After 56 seconds of pumping, neither the ¬is full fault nor the ¬timeout fault has any observable effect on the system. After the next step, the tank is nearly filled, so the sensor should report the tank to be full. The sensor's ¬is full fault might prevent it from doing so, however, so the activation of ¬is full has the observable effect of continued pumping even though the pump should have been deactivated. Consequently, there are two possible scenarios: Either the sensor works, pumping is stopped, and the tank's pressure level decreases, or pumping continues and the timer might prompt the controller to stop pumping. If the timer's ¬timeout fault is also activated, pumping is never stopped and the tank ruptures. Compared to the Kripke structure of the case study's nominal behavior in Figure 5.2, the Kripke structure including the off-nominal behavior is much larger due to its higher number of states and transitions as well as its additional labels that indicate which faults are active in a state. This is obviously to be expected, as it is the very nature of faults to introduce additional behavior and to allow states to be reached that are otherwise unreachable, such as the state in Figure 5.3a where the tank is ruptured. Kripke structures including off-nominal behavior are usually also highly nondeterministic, which again is not particularly surprising due to the nondeterministic nature of fault activations.

New states and transitions are therefore absolutely necessary to model faults, their persistencies, and their effects. For example, there must be a new state that represents a tank rupture and it must be possible to reach that state via at least one new transition when both the pressure sensor fails and the timeout is not triggered. However, the huge number of superfluous additional states and transitions is problematic for formal analysis techniques, significantly reducing their efficiency; e.g., there are three additional states



(a) The Kripke structure resulting from state-based fault modeling has a plethora of redundant states and transitions where faults are active without any observable effects. In particular, both faults can unnecessarily switch to their active states in many situations where they have no effect and hence cannot be activated.



(b) The activation-minimal version of the Kripke structure has no redundant states or transitions by focusing on relevant active faults. The states labeled with f are required to encode the fault's permanent persistency as the model executes multiple fill cycles.

Figure 5.3. Partial view of the Kripke structures for the pressure tank case study including off-nominal behavior that shows the potential reductions in state and transition counts between state-based (left) and activation-minimal (right) fault modeling. The states and transitions belonging to the nominal behavior are dashed whereas the off-nominal behavior is represented by solid states and transitions. States are labeled with P when the pump is running; the number represents both the tank's pressure level and the timer's counter. The faults ¬is full and ¬timeout are active in states that are marked with labels f and t, respectively. For reasons of presentational clarity, the other two faults are omitted and ¬is full and ¬timeout are assumed to be permanent.

for each state of the nominal behavior in Figure 5.2 as well as many transitions between them that do not provide any significant information during safety analyses. The Kripke structure shown in Figure 5.3b can be seen as an activation-minimal abstraction of the one in Figure 5.3a. It is minimal in the sense that irrelevant active faults are omitted while all system states remain reachable, including, in particular, the hazard. The notion of minimality is based on the observation that the exact points in time in which faults become active are irrelevant as long as they do so when they can affect the system. States that are equivalent modulo active faults can thus be unified, thereby reducing both the state and the transition counts significantly. However, the states where ¬is full is active cannot be unified due to the cyclic nature of the model and the fact that the fault is assumed to have permanent persistency: Once ¬is full was activated in fill cycle n, its permanent persistency forces it to remain active in all later fill cycles n' with $n' > n$. Consequently, all states in which the tank is being filled or depleted are duplicated to correctly encode ¬is full's persistency.

VECS, Compass, FSAP/NuSMV, xSAP, and other safety analysis tools [24, 27, 65, 122, 135] share this common, state-based fault modeling approach. All of the redundant states and transitions in Figure 5.3a are artifacts of the way in which these tools generate the Kripke structures from their high-level models during model checking: Nominal and off-nominal behavior is modeled separately and combined together using some product

automaton construction [80]. The off-nominal parts of the model describe the faults' effects as well as their persistencies; during transformation, they introduce the expected additional states and new transitions to represent the off-nominal behavior at the level of Kripke structures. The product automaton construction, however, is what bloats the generated Kripke structure, resulting in many superfluous states and transitions that are irrelevant for the purposes of safety analysis: Each previously existing state of the nominal system behavior is duplicated for all combinations of active faults with various transitions between them. As faults usually do not always cause errors, many of the "activations" that are introduced into the generated Kripke structure in this way are irrelevant as illustrated by Figure 5.3.

In the high-level models, persistency of a fault is typically modeled using a state machine like one of those shown in Figure 3.21 on page 50. Consequently, all other states of the generated Kripke structure have to be combined with the fault's dormant and active states and all previously existing transitions must be duplicated to capture fault activations, deactivations, as well as situations in which the fault remains active or dormant. Transient faults represent the worst case as they occur completely nondeterministically: n additional transient faults increase the generated Kripke structure's reachable state space by a factor of 2^n and each state has an additional 2^n successor states due to the product automaton construction. Permanent faults, by contrast, have an overall lower number of possible successor states compared to transient faults, so the amount of reachable states and transitions might not increase as noticeably; model checking and safety analysis efficiency is reduced significantly with each additional fault in both cases. In Figure 5.3a, for instance, only two of the four faults are considered as otherwise there would have been 15 additional states instead of only 3 for each pressure level within the tank. Moreover, the faults are assumed to be permanent as otherwise all four states of each pressure level with $p < 58$ would have transitions to all four states at pressure level $p + 1$, making a total of 16 transitions per level instead of 9. Obviously, the differences between these numbers increase the more faults are injected into a model; similarly, the benefits of minimization illustrated by Figure 5.3b also increase with the number of faults, reducing the state and transition counts by exponential factors depending on the structure of the model and the effects of the injected faults.

5.1.2 Fault-Aware Kripke Structures

Fault-aware modeling and specification is a fundamental change of model-based safety analysis that inherently considers only activation-minimal Kripke structures similar to the one shown in Figure 5.3b. Fault awareness emphasizes fault activations, making them central to the models of safety-critical systems as well as the safety analysis techniques. The state-based fault modeling approach, by contrast, is primarily concerned with the activation states of the faults, tracking for each fault whether it is active or dormant. Fault-aware modeling can thus be seen as an event-based alternative to state-based fault modeling where events are fault activations and deactivations; however, only the former are considered in the following as there are not many use cases that can take advantage of the latter. The significant state and transition count reductions demonstrated by Figure 5.3b make the potential advantages of fault-aware

modeling evident, decreasing the complexity of the models and thereby increasing comprehensibility as well as, in particular, model checking and safety analysis efficiency as evaluated in Sections 5.4 and 5.5. Additionally, fault awareness introduces formal domains and notations for faults and fault activations, simplifying the formal definitions of fault injection, fault removal, fault-aware LTL, and DCCA in Sections 5.2, 5.3 and 5.5.

Fault-aware Kripke structures amend classical Kripke structures [41, 42] in order to make the models of safety-critical systems aware of the faults they contain and the circumstances under which faults are activated. For safety analysis purposes, only actual fault activations are relevant as they cause some off-nominal behavior that would not have occurred without the activations. In the state-based fault modeling approach, by contrast, fault "activations" without any effects introduce additional states that only differ in the active and dormant faults. Fault awareness thus prevents model checkers from considering such situations with irrelevant active faults during analyses.

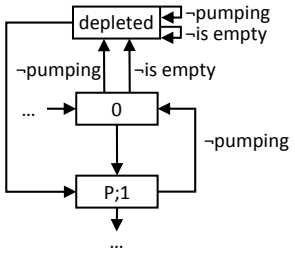
Formal Definition. Fault-aware Kripke structures explicitly denote the faults that can affect the system they represent. They highlight states in which faults can be activated by labeling their outgoing transitions with sets of activated faults. In other words, fault-aware Kripke structures label their transitions with the faults that their transitions activate, allowing the model checker to reason about these activations. This modification can be seen as a mixture between regular Kripke structures and the notion of transition system used by Baier and Katoen [13], the latter allowing arbitrary transition actions in addition to state labels. Formally, fault-aware Kripke structures are defined as follows:

Definition 5.1 (Fault-Aware Kripke Structures)**.** *A* fault-aware Kripke structure K *is represented by the tuple* (P, F, S, R, L, I) *consisting of*

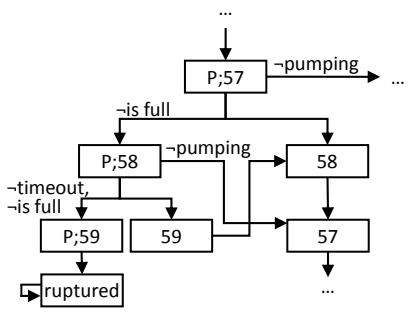
- *a set of atomic propositions* P*,*
- *a set of faults* F*,*
- *a set of states* S*,*
- *a transition relation* $R \subseteq S \times 2^F \times S$ *labeled with fault activations that is*
  - left-total*, i.e.,* $\forall s \in S \,.\, \exists s' \in S, \Gamma \subseteq F \,.\, (s, \Gamma, s') \in R$ *and*
  - activation-minimal*, i.e.,* $(s_1, \Gamma, s_2) \in R \wedge (s_1, \Gamma', s_2') \in R \wedge \Gamma \subsetneq \Gamma' \rightarrow s_2 \neq s_2'$*,*
- *a labeling function* $L : S \rightarrow 2^P$ *indicating the set of atomic propositions holding in a state, and*
- *a non-empty set of initial fault activations and states* $\emptyset \neq I \subseteq 2^F \times S$ *that is* activation-minimal*, i.e.,* $(\Gamma_1, s_1) \in I \wedge (\Gamma_2, s_2) \in I \wedge \Gamma_1 \subsetneq \Gamma_2 \rightarrow s_1 \neq s_2$*.*

By convention, $P(K)$ *stands for* P*,* $F(K)$ *for* F*, and so on. A fault-aware Kripke structure* K *is* finite *if* $P(K)$*,* $F(K)$*, and* $S(K)$ *are finite.*

In the following, the term fault-aware is occasionally omitted, in which case Kripke structure always refers to the fault-aware variant; if the distinction between fault-aware and classical Kripke structures is important, the kind of Kripke structure is explicitly specified. Additionally, only finite Kripke structures are considered. Fault-aware Kripke structures are by definition required to have at least one initial state, that is, $I \neq \emptyset$. Furthermore, the initial states I and the transition relation R must be activation-minimal; the latter must also be left-total. Left totality ensures that for all $s \in S$, there is at least one outgoing transition $(s, \Gamma, s') \in R$. Activation minimality guarantees that two



(a) When the pressure level is or reaches zero again, the sensor should report the tank to be empty in order to start the pump. Activations of either ¬pumping or ¬is empty potentially prevent the system from doing so, resulting in the hazard of a full tank depletion.



(b) The fault-aware Kripke structure corresponding to the activation-minimal classical one in Figure 5.3b. Fault persistency is not encoded into fault-aware Kripke structures, hence the removal of some additional states and transitions compared to the Kripke structure in Figure 5.3b.

Figure 5.4. Parts of the pressure tank case study's fault-aware Kripke structure that explicitly labels its transitions with the minimal amount of possible fault activations to avoid any state or transition redundancy. State label P indicates that the pump is running; the number represents the pressure level within the tank. In contrast to Figure 5.3, all four faults of the case study are considered; if a state has no outgoing transitions labeled with any faults, no fault activations are possible in that state. Transitions are annotated with the faults they activate using a shorthand notation without set braces, that is, ¬is full means $\{\neg\text{is full}\}$ and no label at all abbreviates $\emptyset$.

transitions between the same source and target states are labeled with the minimal causes for the transition, i.e., the minimal sets of fault activations such that no set subsumes any others; for initial states, activation minimality ensures that no initial states exist that can be reached through superfluous fault activations. The benefits of fault-aware Kripke structures for formal safety analysis mainly result from the activation minimality of the transition relations, whereas the other two requirements of left totality and non-empty initial sets are not strictly necessary, they just prevent the Kripke structures from running into deadlocks. The requirement of deadlock freedom, however, does not pose any severe limitations in practice: Firstly, an empty set of initial states is trivial to check for and of little practical use. Secondly, hypothetical deadlock states s can always be made deadlock-free by adding a fresh state s' and transitions $(s, \emptyset, s')$ and $(s', \emptyset, s')$. Thirdly, model checkers are typically equipped to detect deadlocks [38, 98, 116]. Deadlock freedom is therefore only required to simplify the formalization, avoiding some special cases that otherwise would have to be considered later on.

Fault-Aware Model of the Pressure Tank Case Study. Figure 5.4 shows two parts of the fault-aware Kripke structure for the pressure tank case study. Due to activation minimality, there are no transitions labeled with ¬is full or ¬timeout between the states shown in Figure 5.4a as these two faults cannot be activated when the tank is empty. Conversely, the fault ¬is empty is irrelevant in situations where the tank is not empty as seen in Figure 5.4b, whereas an activation of ¬pumping is unable to prevent a tank rupture once activations of both ¬is full and ¬timeout maneuvered the system into a situation in which the tank is fully filled. Transitions labeled with fault activations represent off-

nominal system behavior, whereas unlabeled transitions, or rather, transitions labeled with the empty set, are either taken in error-free contexts or propagate previously introduced errors through the system without any additional fault activations.

The actual state and transition count reductions made possible by activation minimality depend on how often a fault can be activated: ¬timeout, for instance, is only activatable right before the hazard occurs, resulting in a significant state space reduction compared to the state-based fault modeling approach; ¬pumping, by contrast, is activatable in roughly 50% of all states, namely in all states in which the pump is active, and therefore introduces less non-minimal states and transitions in the state-based approach. In contrast to state-based fault modeling, however, there is no duplication of states: Fault-aware Kripke structures do not encode fault persistency; instead, fault persistency is determined during analyses via fault-aware LTL formulas as discussed in Section 5.3.

5.1.3 Semantics and Path Equivalence

The semantics of fault-aware Kripke structures is defined by the set of paths through them. In contrast to the paths of classical Kripke structures [41], the paths of fault-aware ones not only consist of sequences of states but also of the fault activations that cause the transitions between the states. Consequently, a path clearly describes one possible behavior of a system and the circumstances of the fault activations that lead to off-nominal behavior. As soon as a Kripke structure is nondeterministic, which is usually the case once faults are considered, there is more than one path; the set of all such paths describes all possible behaviors of the Kripke structure. Before the notion of paths of fault-aware Kripke structures is defined, however, the more fundamental concept of path fragments is introduced first, representing finite or infinite behavioral sequences of a Kripke structure:

Definition 5.2 (Path Fragments)**.** *A* finite path fragment $\Gamma_0 s_0 \Gamma_1 s_1 \dots \Gamma_n s_n$ *of* length n *and an* infinite path fragment $\Gamma_0 s_0 \Gamma_1 s_1 \dots$ *of a fault-aware Kripke structure K are alternating sequences of fault activations $\Gamma_i \subseteq F(K)$ and states $s_i \in S(K)$ such that $(s_i, \Gamma_{i+1}, s_{i+1}) \in R(K)$ for all $i \geq 0$.*

For a given finite or infinite path fragment ς, $\varsigma[i]$ selects the path's i-th fault-activated state, that is, $\varsigma[i] = (\Gamma_i, s_i)$. When only the activated faults or the state is of interest, the shorthand notations $\varsigma_{\mathrm{F}}[i] = \Gamma_i$ and $\varsigma_{\mathrm{S}}[i] = s_i$ are used, respectively. $\varsigma[..i]$ denotes the finite prefix of a finite or infinite path fragment ς with length i, where $i \leq n$ for finite path fragments of length n. $\varsigma[i..]$ denotes the infinite suffix of an infinite path fragment ς starting at the i-th fault-activated state, that is, $\varsigma[i..] = \Gamma_i s_i \Gamma_{i+1} s_{i+1} \dots$. The concatenation of a finite path fragment $\varsigma = \Gamma_0 s_0 \dots \Gamma_n s_n$ and some infinite path fragment ς' is written as $\varsigma\varsigma' = \Gamma_0 s_0 \dots \Gamma_n s_n \varsigma'[1..]$ if $\varsigma[n] = \varsigma'[0]$.

Definition 5.3 (Paths)**.** *An infinite path fragment ς of a fault-aware Kripke structure K is a* path *of K if $\varsigma[0] \in I(K)$. $paths(K)$ denotes the set of all paths ς of K.*

As Kripke structures are required to be deadlock-free, $paths(K)$ is always non-empty and there can never be such a thing as a finite path. In particular, all finite path fragments ς with $\varsigma[0] \in I(K)$ can be extended into infinity, i.e., there is a path $\varsigma' \in paths(K)$ with $\varsigma'[..n] = \varsigma$ for some $n \in \mathbb{N}$. A state $s \in S(K)$ is reachable if there is some path

$\varsigma \in paths(K)$ that eventually transitions to s, regardless of potential fault activations; trivially, all initial states of a fault-aware Kripke structure are reachable:

Definition 5.4 (Reached & Reachable States)**.** *The set of* reached states $\mathcal{R}(\varsigma)$ *of an infinite path fragment* ς *of a fault-aware Kripke structure* K *is given by* $\{\varsigma_S[i] \mid i \geq 0\}$. *The set of* reachable states $\mathcal{R}(K)$ *of* K *is given by* $\bigcup_{\varsigma \in paths(K)} \mathcal{R}(\varsigma)$.

Similarly, a fault $f \in F(K)$ is considered to be activatable if there is a path $\varsigma \in paths(K)$ on which f is activated at some point:

Definition 5.5 (Activated & Activatable Faults)**.** *The set of* activated faults $\mathcal{A}(\varsigma)$ *of an infinite path fragment* ς *of a fault-aware Kripke structure* K *is given by* $\bigcup_{i \geq 0} \varsigma_F[i]$. *The set of* activatable faults $\mathcal{A}(K)$ *of* K *is given by* $\bigcup_{\varsigma \in paths(K)} \mathcal{A}(\varsigma)$.

Faults $f \in F(K)$ that are not activatable are just as unnecessary as states $s \in S(K)$ that are unreachable: Both can be removed from a Kripke structure without changing its behavior. In order to compare the behavior of two fault-aware Kripke structures, the following notion of path equivalence modulo faults Γ is introduced. It checks whether two Kripke structures have identical paths except for the potentially empty fault set Γ, that is, both Kripke structures must have the same activatable faults except for those faults in Γ, and the paths not activating any faults $f \in \Gamma$ must be the same:

Definition 5.6 (Path Equivalence)**.** *Two fault-aware Kripke structures* K_1 *and* K_2 *are* path-equivalent modulo faults Γ*, denoted as* $K_1 \equiv_\Gamma K_2$*, if for all infinite path fragments* $\varsigma = \Gamma_0 s_0 \Gamma_1 s_1 \ldots$ *with* $s_i \in S(K_1) \cup S(K_2)$, $\Gamma_i \subseteq F(K_1) \cup F(K_2)$*, and* $\Gamma_i \cap \Gamma = \emptyset$ *for all* $i \geq 0$, $\varsigma \in paths(K_1)$ *if and only if* $\varsigma \in paths(K_2)$. $K_1 \equiv_\emptyset K_2$ *is abbreviated as* $K_1 \equiv K_2$.

Kripke structures K_1 and K_2 can only be path-equivalent when their reachable states $\mathcal{R}(K_1)$ and $\mathcal{R}(K_2)$ lie within the intersection of states $S(K_1) \cap S(K_2)$. Additionally, the activatable faults modulo Γ, that is, $\mathcal{A}(K_1) \setminus \Gamma$ and $\mathcal{A}(K_2) \setminus \Gamma$, must lie within the intersection of the fault sets $F(K_1) \cap F(K_2)$. If both prerequisites are satisfied, both Kripke structures must have the exact same paths in order to be considered path-equivalent, optionally disregarding all paths that activate one or more faults $f \in \Gamma$. If $\Gamma = \emptyset$, the entire set of paths of the Kripke structures that are compared must be the same as there are no faults $f \in \Gamma$ that are disregarded.

5.2 Formal Fault Modeling

The systematic fault modeling approach discussed in Chapter 3 establishes the conceptual foundation on which nominal system behavior is conservatively extended through fault injection to incorporate off-nominal behavior. Fault injection mainly consists of two orthogonal parts: The persistency of the fault must be described and its effects on the affected components must be modeled. In accordance with the guidelines for systematic fault modeling, these two aspects are modeled in a structured and modular way in SysML or S# models. Fault-aware Kripke structures, on the other hand, are neither structured nor modular, hence fault modeling at this level is more obscure in the sense that the effects of a fault are modeled by additional transitions labeled with a non-empty set of fault activations; what part of the system is affected by these activations cannot

be easily deduced from fault-aware Kripke structures. Fault persistency, on the other hand, is not modeled at all. Instead, a fault is activated whenever a transition is taken that is labeled with it, whether these activations correspond to the fault's persistency is checked during analysis. Persistency is thus a requirement on the paths of a fault-aware Kripke structure as discussed in Section 5.3.

At the level of fault-aware Kripke structures, the informally introduced concept of fault injection can be formalized to precisely define its meaning and to explore the range of changes that are allowed to be made to a model during fault injection. Additionally, there is the dual of fault injection: Fault removal removes off-nominal behavior from a fault-aware Kripke structure. However, some requirements must be met by the removed faults in order to guarantee deadlock freedom of the resulting fault-aware Kripke structures. The combination of fault injection and fault removal facilitates the new DCCA variant introduced in Section 5.5.

5.2.1 Fault Injection

The nominal behavior of a safety-critical system is commonly modeled first in order to evaluate the modeled system's soundness and adequacy. Subsequently in a separate step, faults are injected into the model to describe the system's off-nominal behavior, enabling formal safety analyses [24, 27, 80]. The kind of changes that are allowed to be made to a model during fault injection are clearly restricted by high-level modeling approaches such as the S# framework or the Compass toolset [82, 151]. Similar restrictions cannot be made for fault-aware Kripke structures, however, due to their unstructured nature. Instead, it is necessary to define a fault injection operation $K \triangleleft F$ that injects the fault set F into a fault-aware Kripke structure K. As the actual effects of the injected faults are highly model-specific, the injection operation cannot yield a single extended Kripke structure but rather describes a set of resulting models incorporating the injected off-nominal behavior. $K \triangleleft F$ is therefore an abstraction of the notion of fault injection where the precise details of the injected off-nominal behavior are irrelevant; each $K' \in K \triangleleft F$ therefore represents one possible injection of faults F into Kripke structure K.

Definition 5.7 (Fault Injection)**.** Injecting *the faults F into a fault-aware Kripke structure K yields the set of* extended *fault-aware Kripke structures $K \triangleleft F$, where for all $K' \in K \triangleleft F$,*

- $P(K') \supseteq P(K)$,
- $F(K') = F(K) \cup F$,
- $S(K') \supseteq S(K)$,
- $R(K') \supseteq R(K)$ *such that for all* $(s, \Gamma, s') \in R(K') \setminus R(K)$, $s \in S \to \Gamma \cap F \neq \emptyset$,
- $L(K')(s) \supseteq L(K)(s)$ *for all* $s \in S(K)$, *and*
- $I(K') \supseteq I(K)$ *such that for all* $(\Gamma, s) \in I(K') \setminus I(K)$, $\Gamma \cap F \neq \emptyset$.

As fault injection extends the nominal behavior with off-nominal behavior, additional propositions, faults, states, labels, transitions, and initial states can be added. However, the original fault-aware Kripke structure must remain intact, i.e., it must be fully embedded into all extended Kripke structures, making it impossible to remove any previously existing parts. In other words, fault injection may add new off-nominal behavior, but

it can never remove previously existing nominal behavior. These restrictions on the allowed changes during fault injection are not required to conduct safety analyses from a formal point of view, they are only necessary for reasons of adequacy: The nominal behavior is carefully modeled and checked for soundness and adequacy, so subsequent fault injections for formal safety analyses should not invalidate any of the previous analyses of the nominal behavior. Despite its abstractness, the formal definition of fault injection is sufficient to formalize and prove the notion conservative extension, that is, it is possible to show that the original Kripke structure and all possible extensions are path-equivalent as long as the injected faults are never activated:

Proposition 5.1 (Conservative Extension)**.** *For a fault set F and fault-aware Kripke structures K and $K_F \in K \triangleleft F$, $K \equiv_F K_F$.*

Proof. "$\Rightarrow$": We have to show that $\varsigma \in \mathit{paths}(K)$ implies $\varsigma \in \mathit{paths}(K_F)$. For a contradiction, assume that i is the position of the first fault-activated state in ς for which K_F cannot continue. If $i = 0$, $\varsigma[0] \in I(K)$ and thus $\varsigma[0] \in I(K_F)$ due to $I(K) \subseteq I(K_F)$, a contradiction. If $i > 0$, $(\varsigma_S[i], \varsigma_F[i+1], \varsigma_S[i+1]) \in R(K)$ and from $R(K) \subseteq R(K_F)$ follows that $(\varsigma_S[i], \varsigma_F[i+1], \varsigma_S[i+1]) \in R(K_F)$, again a contradiction.

"$\Leftarrow$": We have to show that $\varsigma \in \mathit{paths}(K_F)$ with $\mathcal{A}(\varsigma) \cap F = \emptyset$ implies $\varsigma \in \mathit{paths}(K)$. For a contradiction, assume that i is the position of the first fault-activated state in ς for which K cannot continue. If $i = 0$, $\varsigma[0] \in I(K_F) \setminus I(K)$ and thus $\varsigma_F[0] \cap F \neq \emptyset$, which is a contradiction. If $i > 0$, $\varsigma_S[i] \in S(K)$ and $(\varsigma_S[i], \varsigma_F[i+1], \varsigma_S[i+1]) \in R(K_F) \setminus R(K)$, hence $\varsigma_F[i] \cap F \neq \emptyset$, which again yields a contradiction. □

Conservative extension shows that the formal definition of fault injection is purely additive, as previously existing nominal behavior is still exposed by the extended Kripke structures. Newly injected off-nominal behavior can only be reached by activations of injected faults; until then, the extended models behave like the original one. Consequently, all fault injections that adhere to the limitations imposed by Definition 5.7 are guaranteed to be adequate concerning the aspect of conservative extension; whether they adequately describe the off-nominal behavior of a system is outside the scope of the formal techniques. In particular, many $K' \in K \triangleleft F$ are conservative extensions of K but are still practically useless due to improper effect modeling.

Once an activation causes the original part of the Kripke structure to be left, it might potentially be left forever, or the model eventually returns to its nominal behavior. For instance, the fault-aware Kripke structure of the pressure tank case study partially shown in Figure 5.4 only reaches the ruptured state when both faults ¬is full and ¬timeout are activated; once the tank has ruptured, it is never repaired, thus the model never reaches its nominal behavior again. Activations of ¬is empty, on the other hand, can result in the hazard of a complete tank depletion; the Kripke structure is able to reach its nominal behavior again if neither ¬is empty nor ¬pumping are activated.

Comparison to State-Based Fault Injection. State-based fault modeling requires additional states, labels, and transitions for injected faults just to distinguish between the faults' active and dormant states. Additionally, the classical Kripke structures have to incorporate the faults' effects on the system. For fault-aware Kripke structures, however, injecting a fault can only add new transitions when the fault is actually activated;

additional states, labels, and transitions are only required to model the faults' effects on the system. In particular, fault-aware Kripke structures are not concerned with fault persistency, leaving the original behavior completely unmodified. For classical Kripke structures, by contrast, the product automaton construction of the high-level models that incorporates the fault persistency into the extended model [80] also changes the nominal parts of the Kripke structure even though they otherwise have to remain unchanged to ensure conservative extension.

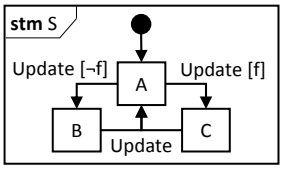
There is another modeling artifact resulting from the state-based fault modeling approach that does not pester fault-aware Kripke structures: With the classical approach, it is always necessary to explicitly forbid previously existing transitions to be taken whenever new transitions are added during fault injection [80, 160]. If the previous behavior is not restricted in such a way, fault "activations" would only nondeterministically have an effect on the system, which would mix up the two orthogonal aspects of fault effects and fault persistency. With fault-aware modeling and specification, such a mix-up cannot happen because effects are described in fault-aware Kripke structures, whereas persistency is described in fault-aware LTL formulas. Figure 5.5 illustrates this difference between both fault modeling approaches.

5.2.2 Fault Removal

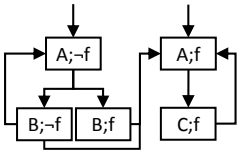
Fault injection is a creative activity in the sense that appropriate fault persistencies must be chosen and the faults' local effects must be modeled, typically using a high-level modeling language for safety-critical systems. The actual fault injection process can subsequently be carried out automatically by a tool like the S# framework or the Compass toolset in order to generate the extended fault-aware Kripke structure including both nominal and off-nominal behavior. As discussed in Sections 5.3 and 5.5, it is sometimes beneficial to remove some of the injected faults during analysis in order to increase analysis efficiency without compromising soundness or completeness of the formal analyses. In contrast to fault injection, however, fault removal is mechanic and can therefore be carried out automatically by a tool such as the S# framework. All of the information necessary to automatically remove one or more faults from a fault-aware Kripke structure is readily available in the model itself, making it unnecessary to track any metadata during fault injection. Analogously to the fault injection operator $K \triangleleft F$, a fault removal operator $K \setminus\!\setminus F$ is defined that removes the faults $f \in F$ from K. The following definition of the operator leaves some degree of freedom, allowing different tool implementations that generate varying reduced Kripke structures. As all of these reduced Kripke structures can only differ in irrelevant details such as unreachable states or transitions, they are pairwise path-equivalent.

Definition 5.8 (Fault Removal)**.** Removing *the fault set $F \subseteq F(K)$ from a fault-aware Kripke structure K yields the set of* reduced *fault-aware Kripke structures $K \setminus\!\setminus F$, where for all $K' \in K \setminus\!\setminus F$,*

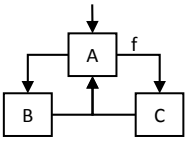
- $P(K') \subseteq P(K)$,
- $F(K') = F(K) \setminus F$,
- $\mathcal{R}(K) \subseteq S(K') \subseteq S(K)$,



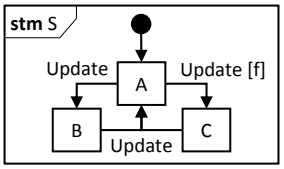
(a) A state machine that deterministically enters state C when the permanent fault f is active.



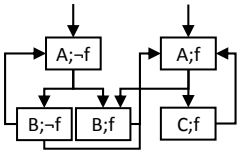
(b) The classical Kripke structure resulting from the state machine to the left.



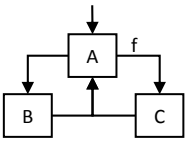
(c) The fault-aware Kripke structure resulting from the state machine to the left.



(d) A state machine that nondeterministically enters state B or C when the permanent fault f is active, mixing up effects and persistency.



(e) The classical Kripke structure for the state machine to the left where an active permanent fault can have no effect.



(f) The fault-aware Kripke structure naturally allows the behavior of the state machine to the left, as persistency is not considered.

Figure 5.5. Comparison of fault-aware and state-based modeling regarding enforcement of fault effects: In the upper row, an activation of fault f in state A deterministically transitions the state machine to state C. The classical Kripke structure in Figure 5.5b and the fault-aware one in Figure 5.5c do the same. In the lower row, by contrast, fault f is not enforced to have any effect, hence the state machine can nondeterministically choose between states B and C, leading to an additional transition in the corresponding classical Kripke structure in Figure 5.5e. Yet, the fault-aware Kripke structure in Figure 5.5f is the same as the one in Figure 5.5c: A transition to state B with the "activation" of fault f is not activation-minimal, hence f has no effect and thus cannot be activated in the first place. In particular, the state machine in Figure 5.5d is nonsensical when f has permanent persistency, as f would be allowed to have no effect even though once it has previously been activated, it is always supposed to have an effect by its very nature. Thus, f would behave like a transient fault even though it is modeled as a permanent one.

- $R(K') = \{(s, \Gamma, s') \in R(K) \mid s, s' \in S(K') \wedge \Gamma \cap F = \emptyset\}$,
- $L(K')(s) = L(K)(s)$ *for all* $s \in S(K')$, *and*
- $I(K') = \{(\Gamma, s) \in I(K) \mid s \in S(K') \wedge \Gamma \cap F = \emptyset\}$.

As fault removal prevents off-nominal behavior, some propositions, faults, states, labels, transitions, and initial states can be removed that would become superfluous after fault removal anyway. However, the original behavior of the fault-aware Kripke structure must remain intact as long as none of the removed faults are activated, i.e., it is impossible to remove any parts that are not related to the removed faults or to add additional parts to the Kripke structure that result in new behavior. In other words, fault removal may remove previously existing off-nominal behavior, but it can never remove previously existing nominal behavior or add any new behavior. Similar to fault injection, these restrictions on the allowed changes during fault removal are not required to conduct formal analyses from a formal point of view, they are only necessary for reasons of adequacy: Fault removal is an optimization that under no circumstances is allowed to influence the analysis results. Despite its abstractness, the formal definition of fault

removal is sufficient to formalize and prove the notion of conservative subtraction, that is, it is possible to show that the original Kripke structure and all possible reduced ones are path-equivalent as long as the removed faults are never activated:

Proposition 5.2 (Conservative Subtraction)**.** *For fault set $F \subseteq F(K)$ and fault-aware Kripke structures K and $K_{\setminus F} \in K \setminus\!\setminus F$, $K \equiv_F K_{\setminus F}$.*

Proof. "⇒": We have to show that $\varsigma \in paths(K)$ with $\mathcal{A}(\varsigma) \cap F = \emptyset$ implies $\varsigma \in paths(K_{\setminus F})$. For a contradiction, assume that i is the position of the first fault-activated state in ς for which $K_{\setminus F}$ cannot continue. If $i = 0$, $\varsigma[0] \notin I(K_{\setminus F})$ and thus either $\varsigma_S[0] \notin S(K_{\setminus F})$ or $\varsigma_F[0] \cap F \neq \emptyset$. The former is impossible as $\varsigma_S[0]$ is reachable in K, the latter is a contradiction. If $i > 0$, $(\varsigma_S[i], \varsigma_F[i+1], \varsigma_S[i+1]) \notin R(K_{\setminus F})$ because either $\varsigma_S[i] \notin S(K_{\setminus F})$, $\varsigma_S[i+1] \notin S(K_{\setminus F})$ or $\varsigma_F[i+1] \cap F \neq \emptyset$. The former two are impossible as both states are reachable in K and the latter is a contradiction.

"⇐": We have to show that $\varsigma \in paths(K_{\setminus F})$ implies $\varsigma \in paths(K)$. For a contradiction, assume that i is the position of the first fault-activated state in ς for which K cannot continue. If $i = 0$, $\varsigma[0] \in I(K_{\setminus F})$ and thus $\varsigma[0] \in I(K)$ due to $I(K_{\setminus F}) \subseteq I(K)$, a contradiction. If $i > 0$, $(\varsigma_S[i], \varsigma_F[i+1], \varsigma_S[i+1]) \in R(K_{\setminus F})$ and from $R(K_{\setminus F}) \subseteq R(K)$ follows that $(\varsigma_S[i], \varsigma_F[i+1], \varsigma_S[i+1]) \in R(K)$, again a contradiction. □

Conservative subtraction guarantees that the formal definition of fault removal is purely subtractive, adding no new behavior while preserving all nominal behavior as well as all off-nominal behavior unrelated to the removed faults. Previously possible off-nominal behavior due to activations of one or more removed faults is no longer reachable. It is possible, and very likely, that some previously reachable states are no longer reachable, or similarly, that some previously activatable faults are no longer activatable even though they are not removed during fault removal. For example, after removing the ¬is full fault from the pressure tank case study's Kripke structure shown in Figure 5.4, the state where the tank is ruptured is no longer reachable and ¬timeout is no longer activatable. All fault removals carried out in accordance with Definition 5.8 are guaranteed not to influence analysis results as they only yield pair-wise path-equivalent Kripke structures:

Corollary 5.1. *For fault set $F \subseteq F(K)$ and fault-aware Kripke structures K and $K_{\setminus F}, K'_{\setminus F} \in K \setminus\!\setminus F$, $K_{\setminus F} \equiv K'_{\setminus F}$.*

Proof. Trivially follows from the fact that there is no choice for the set of initial states, that is, $I(K_{\setminus F}) = I(K'_{\setminus F})$ during fault removal. Moreover, for all states $s \in \mathcal{R}(K)$ originally reachable in K, there is no choice for the successors in the transition relation during fault removal, i.e., $(s, \Gamma, s') \in R(K_{\setminus F})$ if and only if $(s, \Gamma, s') \in R(K'_{\setminus F})$. □

The only differences between two reduced Kripke structures concern unreachable states and unused propositions or state labels. For symbolic model checkers, it is thus beneficial to remove as many unreachable states as possible during fault removal in order to decrease the model checker workload. Explicit-state model checkers, by contrast, inherently do not consider unreachable states, so they in general do not care which of the reduced Kripke structures they analyze from a performance point of view.

In contrast to fault injection, fault removal is an operation that can potentially fail, that is, the set of reduced Kripke structures generated by $K \setminus\!\setminus F$ can be empty: Depending on the faults that are removed, it is possible that the initial states or the transition relations

of all reduced Kripke structures would be empty or non-left-total, respectively, which is explicitly forbidden by the definition of fault-aware Kripke structures. Consequently, if the faults cannot be removed in a way that results in at least one deadlock-free Kripke structure, fault removal fails as no proper reduced Kripke structure exists. A necessary and sufficient condition that guarantees the non-emptiness of $K \setminus\!\setminus F$ is activation independence of the removed fault set. A fault set is activation-independent if activations are never forced, that is, there is always an alternative future in which none of the faults are activated.

Definition 5.9 (Activation Independence)**.** *A fault set $\Gamma \subseteq F(K)$ of a fault-aware Kripke structure K is* activation-independent *in K if there is some $(\Gamma', s') \in I(K)$ such that $\Gamma' \cap \Gamma = \emptyset$ and for all reachable states $s \in \mathcal{R}(K)$ there is a transition $(s, \Gamma'', s'') \in R(K)$ such that $\Gamma'' \cap \Gamma = \emptyset$.*

A fault set is thus considered to be activation-independent when none of its contained faults must be activated in an initial state and all reachable states can branch into at least one future successor state such that the taken transition does not activate any of the faults contained in the set. Fault injection automatically guarantees the first part of activation independence in most modeling and analysis scenarios encountered in practice: Starting from a fault-aware Kripke structure K that does not contain any faults, i.e., $F(K) = \emptyset$, fault injection retains all of the original initial states $I(K)$ that obviously cannot activate any faults. In general, a fault set Γ being activation-independent in a fault-aware Kripke structure K is equivalent to being able to remove Γ from K such that at least one reduced Kripke structure exists:

Theorem 5.1 (Removal of Activation-Independent Faults)**.** *For a fault-aware Kripke structure K and a fault set $F \subseteq F(K)$, F is activation-independent in K if and only if $K \setminus\!\setminus F \neq \emptyset$.*

Proof. "$\Rightarrow$": For a fault-aware Kripke structure K with activation-independent faults $F \subseteq F(K)$, we construct $K_{\setminus F} \in K \setminus\!\setminus F$ such that $P(K_{\setminus F}) = P(K)$, $F(K_{\setminus F}) = F(K) \setminus F, S(K_{\setminus F}) = \mathcal{R}(K), R(K_{\setminus F}) = \{(s, \Gamma, s') \in R(K) \mid s, s' \in S(K_{\setminus F}) \wedge \Gamma \cap F = \emptyset\}$, $L(K_{\setminus F})(s) = L(K)(s)$ for all $s \in S(K_{\setminus F})$, and $I(K_{\setminus F}) = \{(\Gamma, s) \in I(K) \mid s \in S(K_{\setminus F}) \wedge \Gamma \cap F = \emptyset\}$. Indeed, $I(K_{\setminus F}) \neq \emptyset$ as $I(K) \neq \emptyset$ and there is a $(\Gamma, s) \in I(K)$ such that $\Gamma \cap F = \emptyset$ due to the activation independence of F and the reachability of s in K, i.e., $s \in S(K_{\setminus F})$. Moreover, for all $s \in \mathcal{R}(K) = S(K_{\setminus F})$, there is a $(s, \Gamma, s') \in R(K)$ such that $\Gamma \cap F = \emptyset$ due to the activation independence of F. Hence, s and s' are reachable in K, i.e., $s, s' \in S(K_{\setminus F})$, and thus $(s, \Gamma, s') \in R(K_{\setminus F})$. Additionally, $K_{\setminus F}$ is activation-minimal as K is activation-minimal and no new transitions or initial states are introduced during fault removal.

"$\Leftarrow$": Let $K_{\setminus F} \in K \setminus\!\setminus F$ be a reduced Kripke structure for K. For a contradiction, assume that F is not activation-independent in K, so there either is no $(\Gamma, s) \in I(K)$ such that $\Gamma \cap F = \emptyset$ or there is some reachable state $s' \in \mathcal{R}(K)$ such that no transition $(s', \Gamma', s'') \in R(K)$ exists with $\Gamma' \cap F = \emptyset$. In the former case, $I(K_{\setminus F}) = \emptyset$, thus $K_{\setminus F}$ would not be a fault-aware Kripke structure, a contradiction. In the latter case, $s' \in \mathcal{R}(K) \subseteq S(K_{\setminus F})$ but there would be no outgoing transition in $R(K_{\setminus F})$ for s', hence $K_{\setminus F}$ would not be a Kripke structure, again a contradiction. □

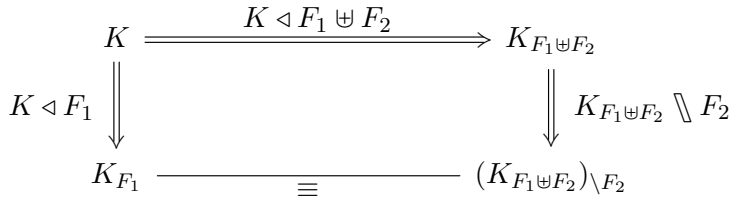


Figure 5.6. Illustration of the relationship between fault injection and fault removal: Starting with a fault-aware Kripke structure K and two disjoint fault sets F_1 and F_2, path-equivalent fault-aware Kripke structures can be obtained by either injecting the relevant faults F_1 only or by first injecting all faults $F_1 \uplus F_2$ and subsequently removing the irrelevant faults F_2.

Due to Theorem 5.1, it is not necessary to check whether a fault set that is to be removed is activation-independent before the set can safely be removed from a fault-aware Kripke structure; if the removal operation succeeds, the removed faults are activation independent in the original Kripke structure. When fault-aware Kripke structures are generated from S# models, activation independence is guaranteed for both transient and permanent faults, thus enabling fault removal.

5.2.3 Removal of Injected Faults

Fault injection and fault removal are two orthogonal operations that can be performed independently on any fault-aware Kripke structure. On the other hand, it is also possible to remove previously injected faults: If some of the injected faults should be suppressed during some analysis activities, their activations could either be ignored by the model checker or they could be removed from the model in the first place, potentially increasing analysis performance. As shown by Proposition 5.2, fault removal preserves all behavior modulo the removed faults, hence such a fault removal optimization does not influence analysis results in any way other than analysis efficiency. However, injecting a potentially large set of faults just to remove most or all of them later on also seems inefficient; especially when DCCAs are conducted, the fault-aware Kripke structures are often checked with only very few relevant faults as discussed in Section 5.5.

Figure 5.6 illustrates that it is in fact irrelevant whether a large set of faults is injected first and a subset of those faults is removed, or whether only the complement of the removed faults is injected in the first place. Regardless of which path is taken through Figure 5.6, it is always possible to construct two path-equivalent fault-aware Kripke structures. Automated analysis tools such as the S# framework are therefore allowed to choose the path that is most convenient and efficient. Formally, the following proposition shows that it is indeed valid to inject only the smaller fault set for analysis instead of injecting all faults and subsequently removing a large portion of them:

Proposition 5.3 (Removal of Injected Faults)**.** *For fault sets F and $F' \subseteq F$ and all fault-aware Kripke structures K, $K_F \in K \triangleleft F$, and $(K_F)_{\setminus F'} \in K_F \setminus\!\setminus F'$, there is a $K_{F \setminus F'} \in K \triangleleft (F \setminus F')$ such that $K_{F \setminus F'} \equiv (K_F)_{\setminus F'}$.*

Proof. By conservative extension we can choose $K_{F \setminus F'} \in K \triangleleft (F \setminus F')$ such that $K_{F \setminus F'} \equiv_{F'} K_F$. Then $K_{F \setminus F'} \equiv (K_F)_{\setminus F'}$ because of Proposition 5.2. □

At the formal level, Proposition 5.3 can only show the existence of path-equivalent Kripke structures as the fault injection operation abstracts from concrete fault effects, yielding sets of extended Kripke structures that represent all conceivable fault effects that could potentially be added to a Kripke structure. The reverse of Proposition 5.3, i.e., the other path through Figure 5.6, trivially holds considering that for all $K_{F_1} \in K \triangleleft F_1$ there is a $K_{F_1 \uplus F_2} \in K \triangleleft F_1 \uplus F_2$ that does simply not add any new transitions or initial states for faults $f \in F_2$ and is otherwise equivalent to K_{F_1}, essentially making the subsequent removal operation unnecessary. For the special case of removing all injected faults, it is possible to formally show that the original Kripke structure K is path-equivalent to all extended and subsequently reduced ones. However, K itself does not result from the injection-removal process as long as at least one new reachable state s is added during fault injection, as s cannot be removed again during fault removal; no reachable states of the extended Kripke structures can be removed.

Proposition 5.4 (Removal of All Injected Faults)**.** *For a fault set F and a fault-aware Kripke structure K, $K \equiv K_{F \setminus F}$ for all $K_F \in K \triangleleft F$ and $K_{F \setminus F} \in K_F \setminus\!\!\setminus F$.*

Proof. Let K_F and $K_{F \setminus F}$ be two arbitrary fault-aware Kripke structures such that $K_F \in K \triangleleft F$ and $K_{F \setminus F} \in K_F \setminus\!\!\setminus F$. $I(K) = I(K_{F \setminus F})$ as $I(K) \subseteq I(K_F)$ and all additional $(\Gamma, s) \in I(K_F) \setminus I(K)$ with $\Gamma \cap F \neq \emptyset$ are removed from $I(K_F)$ during fault removal. Similarly, $R(K) = R(K_{F \setminus F})$ as $R(K) \subseteq R(K_F)$ and all additional $(s, \Gamma, s') \in R(K_F) \setminus R(K)$ with $\Gamma \cap F \neq \emptyset$ are removed from $R(K_F)$ during fault removal. Thus, $K \equiv K_{F \setminus F}$. □

5.3 Formal Properties of Safety-Critical Systems

Formal system specifications describe the properties that a system is expected to exhibit. They are derived from informally stated system requirements in order to formally check that the system model satisfies the requirements. There is a wide variety of specification languages for formalizing system properties, with the class of temporal logics being one possible choice [90]. The following introduces LTL with the standard set of future and past operators [13, 133]. It is amended with explicit support to express actual and potential fault activations in order to formally specify properties over safety-critical systems modeled as fault-aware Kripke structures. Other temporal logics such as Computation Tree Logic (CTL) [13] could also be extended with fault awareness similar to fault-aware LTL.

5.3.1 Fault-Aware Linear Temporal Logic

Fault-aware LTL is an extension of propositional logic that adds temporal modalities for expressing constraints over multiple fault-activated states of infinite paths of fault-aware Kripke structures. For instance, events such as fault activations, hazard occurrences, or the pressure sensor reporting that the tank is full can be expressed using LTL. In contrast to what the adjective "temporal" might suggest, LTL cannot specify the real-time behavior of a system [13, 17]; rather, LTL is based on an abstract time model that refers to the relative order of events only, thereby mimicking the discrete-state, discrete-time nature of Kripke structures [36]. For example, it is possible to express

linear time properties such as "the pressure tank cannot rupture as long as the pressure sensor's ¬is full fault is not activated", which establishes a constraint between a tank rupture and the activation of ¬is full for all paths of a fault-aware Kripke structure. By contrast, it is impossible to specify properties that refer to the real-world timing or duration of events such as "it takes at least one minute of continuous pumping before the tank ruptures".

The following syntax definition of fault-aware LTL formulas lists the propositional logic operators $\neg$ and $\wedge$ only from which all other propositional logic operators such as $\vee$ and $\rightarrow$ can be derived in the usual way [13]. Additionally, it defines two future modalities $\mathbf{X}$ (next) and $\mathbf{U}$ (until) as well as two past modalities $\mathbf{P}$ (previously) and $\mathbf{S}$ (since) with additional modalities defined as usual [13, 133]. Past modalities do not increase the expressiveness of the logic, but can make some formulas exponentially more succinct compared to ones that only use future modalities [139] while in many practical cases still allowing for efficient model checking [18, 170]. Compared to standard definitions of classical LTL with future and past modalities, fault-aware LTL is extended with two new operators related to fault activations: Formula Γ requires that at least the faults in Γ were activated to reach a state, that is, it allows to check whether the Kripke structure reached a state because of the activations of all $f \in \Gamma$, and potentially more, during the last transition. Formula Γ therefore allows a glimpse into the immediate past, whereas the other new operator supported by fault-aware LTL conceptually looks into the immediate future: Formula $\Gamma^?$ checks whether exactly the fault set Γ might potentially be activated when leaving a state, i.e., it allows to check whether precisely the faults $f \in \Gamma$ can be activated to reach the next state. The $\Gamma^?$ operator therefore considers multiple distinct futures that are possible instead of one single future as is usually the case with LTL; the operator is conceptually similar to $\mathbf{EX}$ in CTL. Fault-aware LTL is unable to directly express that a fault is currently active or dormant, a limitation that does not affect analyses of the case studies in any way.

Definition 5.10 (Syntax of Fault-Aware LTL)**.** *Fault-aware LTL formulas Φ are formed over a set P of atomic propositions and a set F of faults according to the following grammar, where φ, φ_1, and φ_2 are fault-aware LTL formulas over P and F, $p \in P$, and $\Gamma \subseteq F$:*

$$\Phi \triangleq \mathrm{true} \mid p \mid \Gamma \mid \Gamma^? \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{X}\,\varphi \mid \varphi_1\,\mathbf{U}\,\varphi_2 \mid \mathbf{P}\,\varphi \mid \varphi_1\,\mathbf{S}\,\varphi_2$$

Other propositional connectives are defined as usual. Additionally, $\varphi_1\,\mathbf{U}^{=}\,\varphi_2$ is defined as $\varphi_1\,\mathbf{U}\,(\varphi_1 \wedge \varphi_2)$, $\mathbf{F}\,\varphi$ (finally) abbreviates $\mathrm{true}\,\mathbf{U}\,\varphi$, and $\mathbf{G}\,\varphi$ (globally) stands for $\neg\,\mathbf{F}\,\neg\varphi$. Analogously, $\mathbf{O}\,\varphi$ (once) abbreviates $\mathrm{true}\,\mathbf{S}\,\varphi$ and $\mathbf{H}\,\varphi$ (historically) stands for $\neg\,\mathbf{O}\,\neg\varphi$.

The semantics of a fault-aware LTL formula is defined over an infinite path fragment ς of a fault-aware Kripke structure K for a given fault-activated state at position i. While an atomic proposition $p \in P(K)$ holds in state $s = \varsigma_{\mathrm{S}}[i]$ of ς when p is in the state's set of labels, that is, $p \in L(K)(s)$, a fault formula Γ with $\Gamma \subseteq F(K)$ holds if state $s = \varsigma_{\mathrm{S}}[i]$ of ς is reached by the activation of at least the faults $f \in \Gamma$, i.e., $\Gamma \subseteq \varsigma_{\mathrm{F}}[i]$. Consequently, there can be additional faults $f \in F(K) \setminus \Gamma$ activated to reach s that are ignored by the formula Γ. The formula's semantics represents the common case without loss of generality, as fault activations are also precisely expressible: State s is reached precisely

because of the activations of faults $\Gamma \subseteq F(K)$, but no others, if $\Gamma \wedge \bigwedge_{f \in F(K) \setminus \Gamma} \neg f$. On the other hand, $\Gamma^?$ with $\Gamma \subseteq F(K)$ holds if state $\varsigma_S[i]$ of ς can be left by a transition that activates precisely the faults $f \in \Gamma$, which fits the common use cases of the operator.

The next state operator $\mathbf{X}\,\varphi$ checks whether φ holds in the next state of path ς, i.e., at $i+1$. Conversely, the previous state operator $\mathbf{P}\,\varphi$ checks whether φ holds in the previous state of path ς, i.e., at $i-1$; if $i = 0$, it does not hold. For instance, the formula $p \rightarrow \mathbf{X}\,f$ with f abbreviating $\{f\}$ checks whether proposition p holding in the current state with index i must subsequently result in the activation of fault f at $i+1$. The finally operator $\mathbf{F}\,\varphi$ and the globally operator $\mathbf{G}\,\varphi$ ensure that φ holds in at least one or in all future states, including the current one, respectively. Formulas $\mathbf{F}\,f$ and $\mathbf{G}\,f$ can therefore be used to check whether fault f will be activated somewhere along the path, or whether it will always be activated on the path. By contrast, the once operator $\mathbf{O}\,\varphi$ and the historically operator $\mathbf{H}\,\varphi$ ensure that φ held in at least one or in all past states, including the current one, respectively. Formula $\mathbf{O}\,f$, for instance, can thus be used to check whether fault f was previously activated at least once. The until operator $\varphi_1\,\mathbf{U}\,\varphi_2$ requires φ_2 to hold at some point on the path, and up to this point, φ_1 must hold. For example, $p\,\mathbf{U}\,f$ requires that eventually fault f is activated and as long as the activation has not happened yet, the proposition p must hold. The since operator is until's dual for the past, meaning that $\varphi_1\,\mathbf{S}\,\varphi_2$ holds if φ_2 held previously or holds now, and since then φ_1 holds. The following definition of the fault-aware LTL semantics is based on the standard semantics of the temporal operators [13, 133]; their semantics are nevertheless defined below for reasons of completeness as they are used extensively in the remainder of this thesis.

Definition 5.11 (Semantics of Fault-Aware LTL)**.** *A fault-aware LTL formula $\varphi \in \Phi$ is* valid *at a position $i \geq 0$ of an infinite path fragment ς of a fault-aware Kripke structure K, written as $\varsigma, i \models \varphi$, if:*

$$
\begin{array}{lll}
\varsigma, i \models \mathrm{true} & & \\
\varsigma, i \models p & \textit{iff} & p \in L(K)(\varsigma_S[i]) \\
\varsigma, i \models \Gamma & \textit{iff} & \Gamma \subseteq \varsigma_F[i] \\
\varsigma, i \models \Gamma^? & \textit{iff} & (\varsigma_S[i], \Gamma, s) \in R(K) \textit{ for some } s \in S(K) \\
\varsigma, i \models \neg\varphi & \textit{iff} & \varsigma, i \not\models \varphi \\
\varsigma, i \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \varsigma, i \models \varphi_1 \textit{ and } \varsigma, i \models \varphi_2 \\
\varsigma, i \models \mathbf{X}\,\varphi & \textit{iff} & \varsigma, i+1 \models \varphi \\
\varsigma, i \models \varphi_1\,\mathbf{U}\,\varphi_2 & \textit{iff} & \textit{there is a } k \geq i \textit{ with } \varsigma, k \models \varphi_2 \textit{ and} \\
 & & \varsigma, j \models \varphi_1 \textit{ for all } i \leq j < k \\
\varsigma, i \models \mathbf{P}\,\varphi & \textit{iff} & i > 0 \textit{ and } \varsigma, i-1 \models \varphi \\
\varsigma, i \models \varphi_1\,\mathbf{S}\,\varphi_2 & \textit{iff} & \textit{there is a } 0 \leq k \leq i \textit{ with } \varsigma, k \models \varphi_2 \textit{ and} \\
 & & \varsigma, j \models \varphi_1 \textit{ for all } k < j \leq i
\end{array}
$$

$\varsigma \models \varphi$ abbreviates $\varsigma, 0 \models \varphi$. φ is valid in K, written as $K \models \varphi$, if and only if $\varsigma \models \varphi$ for all $\varsigma \in paths(K)$.

Due to the implicit universal quantification over all paths ς of K, $\varsigma \models \varphi$ if and only if $\varsigma \not\models \neg\varphi$, but $K \models \neg\varphi$ only implies $K \not\models \varphi$ and not vice versa. It is sometimes difficult or somewhat unintuitive to correctly devise adequate LTL formulas from informal

requirements even though LTL is a comparatively simple temporal logic; μ-calculus, for instance, subsumes LTL and quickly becomes incomprehensible once multiple fix points are nested [54]. It is therefore advisable to consult a standard library of property patterns [21, 49] whenever possible. Such libraries contain various templates, albeit disregarding fault activations, for temporal properties formulated in multiple temporal logics that can be useful to formally specify properties of safety-critical systems.

5.3.2 Fault Suppression and Fault Removal

LTL considers each path of a fault-aware Kripke structure individually, allowing only to get a brief glimpse of other potential paths through the $\Gamma^?$ operator. It is therefore possible to use implications in order to filter out those paths of a Kripke structure that do not satisfy a certain precondition. That is, an LTL formula of the form $(\mathbf{G}\,\varphi_1) \rightarrow \varphi_2$ partitions the set of analyzed paths into those that trivially satisfy the formula due to a violation of φ_1 as well as the remaining paths for which φ_2 must then hold. In particular, LTL formulas of the pattern $(\mathbf{G} \bigwedge_{f\in\Gamma} \neg f) \rightarrow \varphi$ check whether φ holds under the assumption that none of the faults $f \in \Gamma$ are activated, or conversely, the formula suppresses the activations of the faults $f \in \Gamma$ while checking φ. Such formulas are common in plausibility checks that verify certain nominal behavior of a model disregarding all or some of the modeled faults; moreover, DCCA can use these kinds of formulas to check for minimal critical fault sets as explained in Section 5.5.

When checking a formula that suppresses some or all faults $\Gamma \subseteq F(K)$, the paths of Kripke structure K are partitioned into two disjoint sets Σ_s and Σ_a such that $paths(K) = \Sigma_s \uplus \Sigma_a$ with the former containing the paths on which the suppressed faults are never activated and the latter containing the paths that activate some or all of the suppressed faults. Consequently, for all paths $\varsigma \in \Sigma_s$, $\mathcal{A}(\varsigma) \cap \Gamma = \emptyset$ and thus $\varsigma \models \mathbf{G} \bigwedge_{f\in\Gamma} \neg f$, while for the paths $\varsigma \in \Sigma_a$, on the other hand, $\mathcal{A}(\varsigma) \cap \Gamma \neq \emptyset$ and therefore $\varsigma \not\models \mathbf{G} \bigwedge_{f\in\Gamma} \neg f$. Instead of simply discarding Σ_a during model checking, the paths it contains can alternatively be removed from the model in the first place by removing the suppressed faults F from the Kripke structure entirely. The reduced Kripke structure does not contain any paths $\varsigma \in \Sigma_a$, but still retains all other paths $\varsigma \in \Sigma_s$ as shown by the following proposition. Therefore, fault removal at the level of Kripke structures makes it possible to increase analysis efficiency compared to fault suppression with formulas without affecting the outcome of the analyses:

Proposition 5.5 (Fault Suppression)**.** *For a fault-aware Kripke structure K, a fault set $\Gamma \subseteq F(K)$, and an LTL formula $\varphi \in \Phi$ expressible over both K and all $K_{\setminus\Gamma} \in K \setminus\!\setminus \Gamma$, $K \models (\mathbf{G} \bigwedge_{f\in\Gamma} \neg f) \rightarrow \varphi$ if and only if $K_{\setminus\Gamma} \models \varphi$ for all $K_{\setminus\Gamma} \in K \setminus\!\setminus \Gamma$.*

Proof. Let $K_{\setminus\Gamma} \in K \setminus\!\setminus \Gamma$ and $\varsigma \in paths(K)$. If $\varsigma \not\models \mathbf{G} \bigwedge_{f\in\Gamma} \neg f$, $\varsigma \notin paths(K_{\setminus\Gamma})$ due to fault removal as defined in Definition 5.8. Otherwise, $\varsigma \models \varphi$ and hence $K_{\setminus\Gamma} \models \varphi$ as $K \equiv_\Gamma K_{\setminus\Gamma}$ due to Proposition 5.2. □

Proposition 5.5 also shows that all formulas φ holding when faults $f \in \Gamma$ are suppressed hold for all of the reduced Kripke structures resulting from the removal of Γ from the original model. As shown more generally by the following corollary, an LTL formula φ either holds in all reduced Kripke structures or in none of them at all as all of the reduced

models are path-equivalent. Consequently, the concrete choice of the reduced Kripke structure that is checked does not affect the outcome of the analysis, but potentially decreases model checking efficiency when a Kripke structure is chosen with many unreachable states.

Corollary 5.2. *For fault set $\Gamma \subseteq F(K)$, fault-aware Kripke structures K and $K_{\setminus\Gamma}$, $K'_{\setminus\Gamma} \in K \setminus\!\setminus \Gamma$, as well as an LTL formula $\varphi \in \Phi$ expressible over both $K_{\setminus\Gamma}$ and $K'_{\setminus\Gamma}$, $K_{\setminus\Gamma} \models \varphi$ if and only if $K'_{\setminus\Gamma} \models \varphi$.*

Proof. Follows from the fact that $K_{\setminus\Gamma}$ and $K'_{\setminus\Gamma}$ are path-equivalent as shown by Corollary 5.1. Additionally, the labels of all states $s \in \mathcal{R}(K)$ originally reachable in K remain unchanged during fault removal. □

Fault-aware LTL is also able to check whether a fault set $\Gamma \subseteq F(K)$ is activation-independent in K. Due to the equivalence of activation independence and the non-emptiness of the set of reduced Kripke structures $K \setminus\!\setminus \Gamma$, it is thus possible to check whether fault removal succeeds before attempting the operation:

Proposition 5.6 (LTL-Based Characterization of Activation Independence)**.** *A non-empty fault set $\emptyset \neq \Gamma \subseteq F(K)$ is activation-independent in a fault-aware Kripke structure K if and only if $K \not\models \Gamma$ and $K \models \mathbf{G} \bigvee_{\Gamma' \subseteq F(K) \setminus \Gamma} \Gamma'^{?}$.*

Proof. $K \not\models \Gamma$ if and only if there is some $(\Gamma', s') \in I(K)$ such that $\Gamma' \cap \Gamma = \emptyset$ and $\Gamma \neq \emptyset$. Moreover, $K \models \mathbf{G} \bigvee_{\Gamma' \subseteq F(K) \setminus \Gamma} \Gamma'^{?}$ if and only if for all reachable states $s \in \mathcal{R}(K)$ there is a transition $(s, \Gamma'', s'') \in R(K)$ such that $\Gamma'' \cap \Gamma = \emptyset$. □

5.3.3 Persistency Constraints

The orthogonality of fault effects and fault persistency is clearly represented by the fault-aware modeling and specification approach: Effects are modeled in the fault-aware Kripke structures whereas persistency constraints are specified via fault-aware LTL formulas. The former only describe the effects of fault activations, disregarding any persistency constraints. It is therefore possible to use the same fault-aware Kripke structure in multiple analyses with different persistency constraints specified for its faults. The constraints are encoded into the analyzed LTL formulas to filter out paths violating any of them. This separation of concerns reflects the fact that the most important aspect of fault modeling is the adequate injection of fault effects into a model, whereas fault persistency is more of a secondary concern as transient persistency subsumes all other kinds of persistency.

A persistency constraint $\psi \in \Phi$ is a fault-aware LTL formula that specifies the allowed transitions between the active and dormant states of a fault set, that is, it determines the circumstances under which the faults can, cannot, or must be activated. Assuming a set Ψ of persistency constraints that must be satisfied during analyses of a fault-aware LTL formula $\varphi \in \Phi$, $K \models (\bigwedge_{\psi \in \Psi} \psi) \rightarrow \varphi$ is checked instead of $K \models \varphi$ in order to determine the validity of φ in K. Consequently, all paths violating one or more persistency constraints $\psi \in \Psi$ are ignored during analysis, making it possible to choose the kind of persistency of each fault for each analyzed formula individually without requiring any modifications to the underlying fault-aware Kripke structure.

Transient Persistency. Activations of a transient fault $f \in F(K)$ must always be completely nondeterministic: Whenever f can be activated, there must also be a transition that does not activate it and its activation must also not be enforced in all initial states. Otherwise, f's activation would be deterministically enforced at some point, contradicting the informal notion of transient persistency. Consequently, a fault $f \in F(K)$ has transient persistency if it is activation-independent in K. Transient faults do not require any persistency constraints during model checking as long as the faults are activation-independent in the underlying Kripke structure; otherwise, a constraint similar to the one given in Proposition 5.6 must be taken into account. Compared to the state-based fault modeling approach, transient faults do not introduce any additional states at all to model their activation states; they only introduce new states if their activations actually lead to new off-nominal behavior. Due to fault awareness, transient faults thus no longer represent the worst case as with state-based fault modeling, but the best case instead; a significant advantage given that transient faults subsume all other kinds of persistency and are thus most commonly used.

Permanent Persistency. Similar to transient faults, a permanent fault $f \in F(K)$ also requires completely nondeterministic activation, that is, activation independence in K. Additionally, however, a permanent fault also requires that after its first activation, it must always be activated when an activation is again possible. It is therefore necessary to ignore all paths $\varsigma \in \mathit{paths}(K)$ during the analysis of K where an activation of f is not enforced after the first activation has already happened. Using fault-aware LTL, the constraint can be specified as

$$\mathit{perm}(f) \triangleq \mathbf{G}((f^{?} \wedge \mathbf{O}\, f) \rightarrow \mathbf{X}\, f)$$

for a fault $f \in F(K)$: If an activation of f is possible in any state and it has previously been activated at least once, it must also be activated in the next state. Thus, the constraint ensures that the first activation of a permanent fault enforces all subsequently possible activations despite the fault being activation-independent in K; activation independence is only considered for the first activation on a path, all subsequent activations are deterministically enforced by the constraint. In general, however, the persistency constraint for permanent faults is not compositional as illustrated by Figure 5.7 and must therefore be specified globally for all persistent faults $\Gamma \subseteq (K)$ of a fault-aware Kripke structure, basically considering each subset of Γ to be a permanent fault as well:

$$\mathit{perm}(\Gamma) \triangleq \bigwedge_{\emptyset \neq \Gamma' \subseteq \Gamma} \mathbf{G}((\Gamma'^{?} \wedge \bigwedge_{f \in \Gamma'} \mathbf{O}\, f) \rightarrow \mathbf{X}\, \Gamma')$$

Permanent Persistency with Maintenance. Another kind of persistency is introduced by permanent faults that can become dormant again through maintenance, that is, the faults can be repaired. They act like regular permanent faults, except that a maintenance operation repairs the cause for their initial activations, no longer enforcing their activations until they are nondeterministically activated again. For example, the pressure sensor's ¬is full fault of the pressure tank case study could be specified as a permanent fault that can be repaired during regular maintenance intervals. As long as the sensor is broken, ¬is full must always be activated, but once the sensor is repaired, these activations are no longer required; of course, there will be future activations as

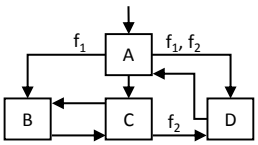


Figure 5.7. A fault-aware Kripke structure with activation-independent fault sets $\{\mathsf{f}_1\}$, $\{\mathsf{f}_2\}$, and $\{\mathsf{f}_1, \mathsf{f}_2\}$ illustrating that the persistency constraints for permanent faults are not compositional: Assuming only $perm(\mathsf{f}_1)$ and $perm(\mathsf{f}_2)$, the finite path fragment $\emptyset\mathsf{A}\mathsf{f}_1\mathsf{B}\emptyset\mathsf{C}\mathsf{f}_2\mathsf{D}\emptyset\mathsf{A}\mathsf{f}_1\mathsf{B}$ would be allowed. The fragment's last transition violates the informal notion of permanent persistency as the previous activations of f_1 and f_2 only allow A's outgoing transition to D that activates both faults simultaneously. However, the individual persistency constraints $perm(\mathsf{f}_1)$ and $perm(\mathsf{f}_2)$ do not enforce the selection of this transition, only the combined constraint $perm(\{\mathsf{f}_1, \mathsf{f}_2\})$ does by considering the entirety of all previous and future activations of all subsets of $\{\mathsf{f}_1, \mathsf{f}_2\}$.

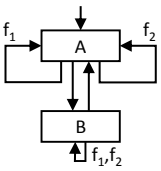
soon as the sensor becomes broken again. Repairable permanent faults act like regular permanent ones within a single maintenance interval, but are more similar to transient ones across multiple maintenance intervals. The situations in which maintenance occurs have to be specified in the analyzed fault-aware Kripke structure through one or more state labels, mapping each fault to its corresponding maintenance label with a function $M\colon F(K) \to P(K)$; M is not required to be injective, as more than one fault can be repaired during maintenance. The persistency constraint for a fault set $\Gamma \subseteq F(K)$ of repairable permanent faults is thus given by

$$perm(\Gamma, M) = \bigwedge_{\emptyset \neq \Gamma' \subseteq \Gamma} \mathbf{G}((\Gamma'^{?} \wedge \bigwedge_{f \in \Gamma'}(\mathbf{O}\, f \wedge \neg(\neg f\, \mathbf{S}\, M(f)))) \rightarrow \mathbf{X}\, \Gamma'),$$

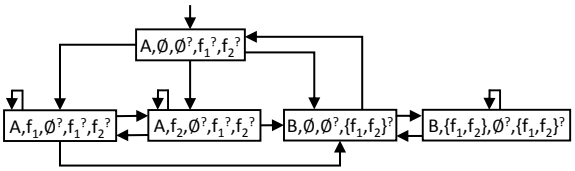
that is: Whenever the faults Γ can be activated, they must be activated in the next state provided that all faults $f \in \Gamma$ have previously been activated at least once since the last maintenance, if any. As for regular permanent faults, the constraint must be globally specified for all repairable permanent faults. In fact, whenever both repairable and non-repairable permanent faults are analyzed, the constraint must not only be used for the repairable ones, but also for all non-repairable ones $\Gamma' \subseteq F(K)$ instead of $perm(\Gamma')$. Otherwise, the same situation as in Figure 5.7 would arise for combinations of different kinds of permanent faults. To adequately use the constraint with non-repairable faults, $M(f)$ must be some proposition $p \in P(K)$ for all $f \in \Gamma'$ such that no state is ever labeled with p, ensuring that no non-repairable $f \in \Gamma'$ can ever be repaired.

5.4 Model Checking Safety-Critical Systems

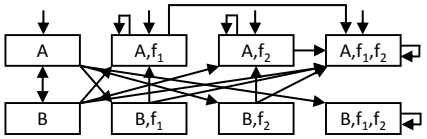
In order to check whether some path through a fault-aware Kripke structure violates a fault-aware LTL formula, a model checker fully automatically explores the Kripke structure in order to unequivocally determine whether the analyzed formula is satisfied or violated. In case of a violation, the model checker additionally yields a (prefix of) a path that explains the circumstances of how the Kripke structure is able to reach a state in which the analyzed formula is violated. Such a counterexample to the formula's validity is useful to determine the causes of the violation, therefore contributing to the understanding and the fix of the problem. As there are no model checkers that directly support fault-aware Kripke structures and fault-aware LTL, the two formalisms must be



(a) A fault-aware Kripke structure with two states and two faults that are only independently activatable in state A and only jointly activatable in state B.



(b) The classical Kripke structure generated from the fault-aware one to the left in accordance with Definition 5.12 requires additional states labeled with actual fault activations f_1,f_2, and $\{f_1, f_2\}$ as well as potential fault activations $f_1{}^?$,$f_2{}^?$, and $\{f_1, f_2\}^?$. As usual, f abbreviates $\{f\}$ while $f^?$ abbreviates $\{f\}^?$.



(c) The classical Kripke structure corresponding to the fault-aware one above using state-based fault encoding under the assumption that both f_1 and f_2 are permanent faults.

Figure 5.8. Illustration of the increase in states and transitions when converting fault-aware Kripke structures to classical ones. The fault-aware Kripke structure has two states and five transitions, whereas the classical one requires five states and thirteen transitions to encode the information about actual and potential fault activations. With state-based fault modeling, on the other hand, the increase is even worse, requiring a total of eight states in order to also encode the activation states of the two faults as well as 18 transitions. If the two faults were transient, the transition count would be even higher.

mapped to their classical counterparts for analysis. The mapping encodes actual and potential fault activations of a fault-aware Kripke structure into atomic propositions Γ and $\Gamma^?$, respectively, to support the transformation of fault-aware LTL; furthermore, all states of the classical Kripke structure are conceptually duplicated for each fault set that can be activated to reach them. Fault-aware LTL formulas can then be mapped in a straightforward way as the formulas Γ and $\Gamma^?$ are simply considered to be propositions; the generated Kripke structure already contains the required state labels to do so. If necessary, past modalities are dealt with in the usual way for model checkers that only support future operators [18, 170].

Definition 5.12. *A fault-aware Kripke structure $K = (P, F, S, R, L, I)$ corresponds to a classical Kripke structure $K' = (P', S', R', L', I')$ where*

- $P' = P \cup 2^F \cup \{\Gamma^? \mid \Gamma \subseteq F\}$,
- $S' = 2^F \times S$,
- $R' = \{((\Gamma, s), (\Gamma', s')) \mid (s, \Gamma', s') \in R\}$,
- $L'(\Gamma, s) = \Gamma \cup L(s) \cup \{\Gamma'^? \mid \exists s' \in S \,.\, (s, \Gamma', s') \in R\}$, *and*
- $I' = I$.

Figure 5.8 illustrates the conversion of a fault-aware Kripke structure to a classical one, contrasting the conversion with a classical Kripke structure using state-based fault modeling. The generated Kripke structure has more states, state labels, and transitions

	State-Based			Fault-Aware			
Case Study	States	Transitions	Time	States	Transitions	Time	
Height Control	249	1,219,724	1.5d	1.3	57.4	14.2s	9127x
Railroad Crossing	116	10,564	12m	2.5	8.5	1.9s	379x
Hemodialysis Machine	13	6,819	42m	0.4	108.1	38.2s	66x

Table 5.1. The results of the S#-based evaluation of the explicit-state efficiency improvements that can be achieved with fault-aware modeling and specification compared to state-based fault modeling. Three of the case studies were evaluated on a 3.4 GHz quad-core CPU: A scaled-down version of the height control system, the radio-controlled railroad crossing, and the hemodialysis machine. The "States" columns show the models' approximate amount of reachable states in millions, "Transitions" the approximate amount of reachable transitions in millions, and "Time" the time it takes to enumerate all states. The last column shows the analysis time speedup that varies greatly depending on the case study. For the height control, analysis efficiency improves by almost four orders of magnitude due to the large number of faults that cannot be activated in many situations. Activation minimality suppresses these redundant activations, leading to significant reductions especially since there are many other sources of nondeterminism in the model. The hemodialysis machine, on the other hand, profits the least with lower reductions of both state and transition counts compared to the other two case studies. Nevertheless, fault awareness still results in a notable analysis time improvement.

than the fault-aware one in order to support fault-aware LTL, in particular its Γ and $\Gamma^?$ operators. However, the additional states and transitions are only reachable when some faults are actually activatable in a state of the fault-aware Kripke structure, whereas state-based fault modeling introduces many additional and often superfluous states and transitions in all situations. Moreover, the transition count increases more noticeably for transient faults than for permanent ones with the state-based approach, whereas fault persistency is not expressed in and therefore irrelevant for fault-aware Kripke structures. In the worst case, the set of reachable states increases by an exponential factor when converting a fault-aware Kripke structure into a classical one; fault-aware modeling and specification therefore is an optimization that is not necessarily better than state-based fault modeling in the general case, that is, it is trivial to construct fault-aware Kripke structures that are just as large as their state-based counterparts.

Explicit-state model checkers check the validity of the analyzed formulas as they explore the states of the analyzed Kripke structure [41]. Their performance is mostly dominated by two factors: The number of states that have to be scanned as well as the number of transitions that have to be explored. Highly nondeterministic models, in particular ones including many fault activations, typically have many outgoing transitions per state, slowing down explicit-state model checkers. Fault-aware modeling and specification reduces state and transition counts through the activation minimality of the fault-aware Kripke structures in order to increase explicit-state analysis efficiency. Table 5.1 shows the analysis time improvements for three of the case studies, comparing the state and transition counts as well as the analysis time reductions of the S# models when either using state-based fault encoding or activation-minimal model execution. At the formal level, these two execution modes correspond to analyses using classical or fault-aware Kripke structures, respectively.

5.5 Deductive Cause Consequence Analysis

DCCA is a fully automated, model checking-based safety analysis technique originally introduced by Ortmeier et al. [159]. In the following, it is redefined over fault-aware Kripke structures using fault-aware LTL with some conceptual improvements that increase its applicability, usability, and efficiency. The following assumes a finite fault-aware Kripke structure K to be given that represents the system to be analyzed for a hazard H, with H specified as a propositional logic formula over K not referencing any faults $f \in F(K)$. By model checking a series of formulas for hazard H and Kripke structure K, DCCA computes all minimal critical fault sets $\Gamma \subseteq F(K)$ that can cause an occurrence of H. DCCA thus uncovers cause consequence relationships between faults (the causes) and hazards (the consequences). Three basic principles of causality [128] underlie DCCA that capture the intuitive notion of what is considered to be a cause for a consequence and what cannot possibly be a cause:

- **Ordering.** The causes identified by DCCA occur strictly before the consequence. A fault that is activated for the first time after the hazard has already occurred cannot reasonably be expected to have caused the hazard. In fact, it is more likely that the activation of the fault is a result of the hazard so that the hazard is the cause for the fault and not vice versa. For instance, a tank rupture in the pressure tank case study could potentially destroy the pump, thus activating the ¬pumping fault; in such a situation, the tank rupture is the cause for the pump failure and not the other way around. Whether a cause must occur strictly before its consequence or whether both can occur simultaneously depends on the modeling formalism and can be debatable. In the state-based fault modeling approach, for instance, original Kripke structures label their states with the faults' activation states. It can thus be somewhat obscure whether a state representing the hazard and making a fault active for the first time is a witness showing that the fault caused the hazard. Fault-aware Kripke structures do not have such ambiguities: Transitions are labeled with fault activations, hence activations conceptually always occur precisely in those situations where they have an effect, i.e., a state representing the hazard that is entered through a transition labeled with a fault activation naturally is a witness for the fault causing the hazard.
- **Traceability.** All causes identified by DCCA are in fact able to cause the consequence in at least one scenario. It is unrealistic to expect that a fault activation always results in a hazard under all circumstances; there might be situations in which the consequence is avoided because of sheer luck or for some systematical reason resulting from the interactions of a system's components. In the pressure tank case study, for example, an activation of ¬is full does not always result in a tank rupture; only under special circumstances, namely when ¬timeout is activated as well, a rupture occurs. However, in order for a fault to be considered a cause for a hazard, there must indeed be at least one situation where its causation of the hazard is traceable; the aforementioned situation in the pressure tank case study is such a witness showing how activations of ¬is full and ¬timeout can lead to tank ruptures.
- **Minimality.** The causes identified by DCCA are minimal as there are no spurious causes that are actually irrelevant for the occurrence of the consequence. Taken to

the extreme, the set of all faults is most likely a cause for a hazard. However, no information can be derived from such a point of view, i.e., it is impossible to devise any additional measures to increase system safety. Therefore, the smallest sets of faults that can lead to a hazard are of primary interest. In the pressure tank case study, the fault set consisting of ¬is full and ¬timeout is one minimal cause for a tank rupture; spurious additional causes such as ¬pumping are not only irrelevant for the occurrence of the hazard, they might even work to prevent the hazard: Activations of ¬pumping can prevent tank ruptures as the tank is never fully filled, potentially even circumventing activations of ¬is full, for instance.

The three principles of causality lead to the notions of potential and effective causes for a consequence [80, 128]. Events such as fault activations are potential causes for a consequence if they adhere to the ordering principle, that is, they must occur before the consequence. Effective causes, on the other hand, are potential causes that also satisfy the other two principles, namely traceability and minimality. Potential causes are therefore hypothesized causes for a consequence whose effectiveness requires further analysis. DCCA is a formal analysis technique that systematically checks all combinations of modeled faults to determine whether they are potential causes. Additionally, it reasons about the implicitly modeled error propagation chains in order to trace fault activations to subsequent hazard occurrences as well as to determine the minimal sets of faults that do so. Once DCCA identified all minimal critical fault sets, all effective causes for the hazard that are related to fault activations are uncovered.

5.5.1 Critical and Safe Fault Sets

Depending on whether a set of faults $\Gamma \subseteq F(K)$ is an effective cause for a hazard H, DCCA classifies Γ as either minimal critical or safe, respectively. The activation of a minimal critical fault set might therefore result in the occurrence of the hazard later on, but the hazard can potentially be avoided under certain circumstances such as changing environmental conditions or additional fault activations that counteract the negative effects of the previously activated ones. Activations of all faults contained in a safe fault set, however, are guaranteed to be unproblematic insofar as they themselves cannot lead to the hazard; of course, the more faults are activated, the more likely it is that additional fault activations lead the system into critical situations. The criticality property of a fault set is formally defined as follows:

Definition 5.13 (Critical Fault Sets)**.** *Let H be a propositional logic formula not referencing any faults $f \in F(K)$. A fault set $\Gamma \subseteq F(K)$ is* critical *for hazard H if and only if there is some $\varsigma \in paths(K)$ with $\varsigma \models only_\Gamma \, \mathbf{U}^{=} H$ and $only_\Gamma \triangleq \bigwedge_{f \in F(K) \setminus \Gamma} \neg f$.*

A fault set Γ is thus critical for a hazard H if there is the possibility that H occurs and up to and including that point, at most the faults in Γ occur. The $\mathbf{U}^{=}$ operator is defined in Definition 5.10 as $\varphi_1 \, \mathbf{U}^{=} \, \varphi_2 \triangleq \varphi_1 \, \mathbf{U} (\varphi_2 \wedge \varphi_1)$, requiring φ_1 to continue to hold even for the state where φ_2 is already satisfied. For critical fault sets, the use of the $\mathbf{U}^{=}$ operator guarantees that the transition leading to the state where the hazard occurs still enables at most the faults $f \in \Gamma$; if $\mathbf{U}$ was used instead, there could be activations of faults $f \in F(K) \setminus \Gamma$ right before the hazard occurs, which would be inadequate.

The criticality property grossly overestimates the impact of a fault set on the modeled system by not even requiring all contained faults to be activated before the hazard occurs; therefore, criticality does not even satisfy the ordering requirement of causality. A critical fault set only is a potential cause for the hazard in the sense that simultaneous or consecutive activations of some of its contained faults can lead to the hazard. Minimal critical fault sets, on the other hand, adhere to all three principles and thus represent effective causes for the analyzed hazard as they exclusively contain relevant faults, that is, if one of the faults is removed form the set, it is no longer critical at all:

Definition 5.14 (Minimal Critical Fault Sets)**.** *A critical fault set* Γ *is* minimal *if no proper subset* $\Gamma' \subsetneq \Gamma$ *is critical.*

The ordering principle is satisfied by minimal critical fault sets even though it is only checked implicitly: For some minimal critical fault set $\Gamma \subseteq F(K)$, there is a path $\varsigma \in paths(K)$ that shows Γ's criticality, albeit without requiring any faults $f \in \Gamma$ to be activated before the analyzed hazard H occurs; instead, the criticality property only checks that all other faults $f \in F(K) \setminus \Gamma$ are not activated before the occurrence of H. As Γ is minimal critical, there are no subsets $\Gamma' \subsetneq \Gamma$ such that Γ' is critical, as otherwise Γ would not have been minimal critical. Consequently, ς is a witness showing how and when all $f \in \Gamma$ must be activated prior to the occurrence of H. Moreover, ς also fulfills the traceability requirement as it clearly shows how the fault set Γ can cause H. The minimality principle is also satisfied by definition, as the removal of a single fault $f \in \Gamma$ from Γ results in $\Gamma \setminus \{f\}$ to no longer be critical at all.

Most model checkers do not support checking whether an LTL formula holds for at least one path instead of checking whether it holds for all paths, making it infeasible to identify critical fault sets directly; a notable exception is SPIN [98]. On the other hand, the dual of critical fault sets, safe fault sets, can be checked using regular LTL:

Definition 5.15 (Safe Fault Sets)**.** *Let* H *be a propositional logic formula not referencing any faults* $f \in F(K)$*. A fault set* $\Gamma \subseteq F(K)$ *is* safe *for hazard* H *if and only if* $K \models \neg(only_\Gamma \, \mathbf{U}^{=} \, H)$ *with* $only_\Gamma$ *as in Definition 5.13.*

A fault set Γ is thus considered safe for a hazard H if it is impossible that at most the activations of faults $f \in \Gamma$ result in an occurrence of H. In some situations, however, a safe fault set can nevertheless be seen as a potential cause for the analyzed hazard, for instance when it is possible to activate some or all of its containing faults before additional fault activations lead to the hazard. But only the activations of faults contained in the set never lead to the hazard, hence safe fault sets are never effective causes. Intuitively, a fault set is critical if and only if it is not safe; formally, this correlation trivially follows from Definitions 5.13 and 5.15 and the semantics of LTL's satisfaction relation from Definition 5.11.

Corollary 5.3 (Equivalence of Safe and Non-Critical Fault Sets)**.** *A fault set* $\Gamma \subseteq F(K)$ *is safe for a hazard* H *if and only if it is not critical for* H*.*

Proof. For a safe fault set $\Gamma \subseteq F(K)$, $K \models \neg(only_\Gamma \, \mathbf{U}^{=} \, H)$ is equivalent to $\varsigma \models \neg(only_\Gamma \, \mathbf{U}^{=} \, H)$ for all $\varsigma \in paths(K)$ due to Definition 5.11. Therefore, there is no $\varsigma \in paths(K)$ such that $\varsigma \models only_\Gamma \, \mathbf{U}^{=} \, H$ by semantic extraction of the negation also in accordance with Definition 5.11. Thus, Γ is not critical and vice versa. □

A complete DCCA for a hazard H checks all combinations of faults $\Gamma \subseteq F(K)$ to determine whether they are safe, thereby implicitly deducing those fault sets that are minimal critical for the analyzed hazard. While safe fault sets are checked mostly due to the limitations of the formal analysis tools that do not support existential LTL over individual paths of a Kripke structure, the minimal critical fault sets are the actual results that are expected from safety analysis techniques: Minimal critical sets are minimal root causes for a hazard, identifying the "simplest" circumstances under which a safety-critical system potentially causes harm. In particular, critical fault sets violate the formula shown in Definition 5.15 and the model checker generates a counterexample, that is, a witness that precisely shows one situation in which the activations of (a subset of) the faults contained in the set can lead to the analyzed hazard.

Definition 5.16 (Deductive Cause Consequence Analysis)**.** *Let H be a propositional logic formula not referencing any faults $f \in F(K)$.* Deductive Cause Consequence Analysis *is the identification of the set of all minimal critical fault sets $\Lambda_H = \{\Gamma \subseteq F(K) \mid \Gamma$ is minimal critical for $H\}$.*

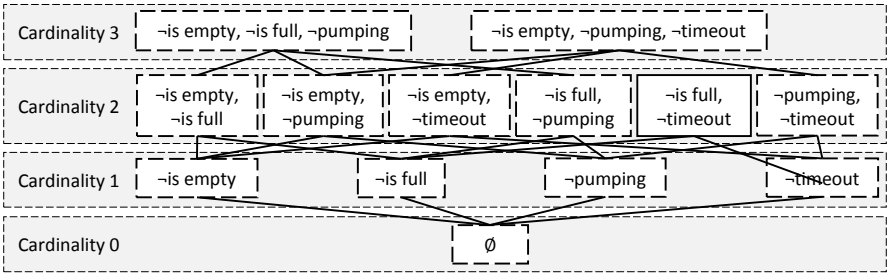
DCCA guarantees that for every minimal critical fault set, there is at least one path in which the hazard is caused precisely by the faults contained in this set, satisfying all three principles of causality. As DCCA identifies all effective causes for a hazard, there can neither be any additional minimal critical fault sets nor can there be any irrelevant ones. In order to compute all minimal critical fault sets for a hazard, however, all $2^{|F(K)|}$ subsets of $F(K)$ have to be checked in the worst case. Such a brute force approach thus requires an effort exponential in the number of analyzed faults. In practice, however, the number of required checks is usually significantly lower, as the criticality property is monotonic with respect to set inclusion: Once a fault set $\Gamma \subseteq F(K)$ is known to be critical, all supersets $\Gamma \subseteq \Gamma' \subseteq F(K)$ are critical as well since additional fault activations cannot be relied upon to fix negative effects of other faults [193]; consequently, more fault activations never make a system safer. Conversely, for all safe fault sets $\Gamma \subseteq F(K)$, all subsets $\Gamma' \subseteq \Gamma$ are safe as well: Less fault activations never make a system less safe. Notably, both claims hold regardless of the modeled system and whether the fault sets $\Gamma \subseteq F(K)$ are activation-independent in K; DCCA never forces their activation, but instead suppresses the activations of the complement set $F(K) \setminus \Gamma$.

Theorem 5.2 (Monotonicity)**.** *For fault sets $\Gamma, \Gamma' \subseteq F(K)$ with $\Gamma \subsetneq \Gamma'$:*

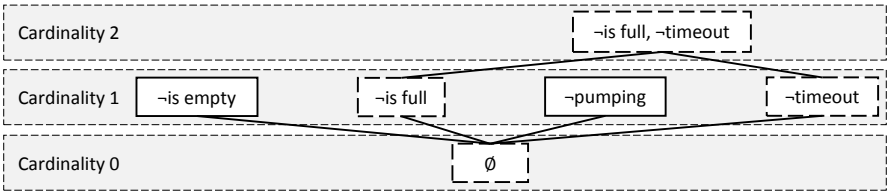
1. *If Γ is critical for a hazard H, so is Γ'.*
2. *If Γ' is safe for a hazard H, so is Γ.*

Proof. The definition of $only_\Gamma$ does not require the faults in Γ to be activated. Consequently, if there is a path $\varsigma \in paths(K)$ such that $\varsigma \models only_\Gamma \mathbf{U}^{=} H$, we also have $\varsigma \models only_{\Gamma'} \mathbf{U}^{=} H$. Conversely, if Γ' is safe then there is no path $\varsigma \in paths(K)$ such that $\varsigma \models only_{\Gamma'} \mathbf{U}^{=} H$, so there can be no such path for Γ either. □

A bottom-up approach taking advantage of monotonicity to conduct DCCAs more efficiently than the naïve brute force way is illustrated in Figure 5.9: All subsets of $F(K)$ can be checked by increasing cardinality, that is, the empty set of faults is checked first, subsequently all sets of cardinality one are checked, and so on. At each cardinality level, all sets that are supersets of some previously found critical fault set are skipped as the



(a) For the hazard of tank ruptures, 13 out of 16 fault sets have to be checked for criticality in order to identify the hazard's only minimal critical fault set $\{\neg\text{is full}, \neg\text{timeout}\}$.



(b) For the hazard of tank depletions, only 6 out of 16 fault sets have to be checked for criticality due to monotonicity in order to identify the hazard's two minimal critical fault sets $\{\neg\text{is empty}\}$ and $\{\neg\text{pumping}\}$.

Figure 5.9. Overview of the fault sets that are checked by Algorithm 2 for the two hazards of the pressure tank case study. Safe fault sets are indicated by a dashed outline, whereas minimal critical ones are solid; non-minimal critical ones are not shown.

outcome of the check is already known: Because of the monotonicity of the criticality property, such fault sets are critical as well, but certainly not minimal critical. With this analysis approach, all critical sets that are found are immediately known to be minimal and no unnecessary checks are carried out. Additionally, minimal critical fault sets are often rather small and are thus found quickly when checking by increasing cardinality as they consist of very few faults only. Algorithm 2 shows an anytime algorithm for conducting DCCAs bottom-up using the monotonicity of criticality; the S# framework can additionally take advantage of the monotonicity of safe fault sets to further reduce the number of required checks as explained in Chapter 6.

Original Definition of Criticality. DCCA was originally introduced using CTL [159]. The following definition is the original one of the criticality property that uses CTL's exists-until quantifier $\mathbf{EU}^{=}$ to check whether for all initial states $(\Gamma, s) \in I(K)$, there is an infinite path fragment ς for K with $\varsigma[0] = (\Gamma, s)$ on which the hazard H occurs and before that, at most the faults $f \in \Gamma$ are activated. CTL could be extended to support fault-aware Kripke structures similar to fault-aware LTL.

Definition 5.17. *Let $|I(K)| = 1$ and H be a propositional logic formula not referencing any faults $f \in F(K)$. A fault set $\Gamma \subseteq F(K)$ is* critical *for hazard H if $K \models only_\Gamma\ \mathbf{EU}^{=} H$ with $only_\Gamma$ as in Definition 5.13.*

As even CTL's exists-until quantifier $\mathbf{EU}^{=}$ is all-quantified over all initial states when it is used as the top-level operator of a CTL formula, criticality thus originally had to assume Kripke structures to have a single initial state only, whereas Definition 5.15 for safe fault sets does not have to make this assumption. For Kripke structures with

function DCCA(K, H, k_{max})
  $\Delta_{check} \leftarrow \{\emptyset\}$
  $\Lambda_H \leftarrow \emptyset$
  $k \leftarrow 0$
  while $\Delta_{check} \neq \emptyset \wedge k \leq k_{max}$ **do**
    $\Delta_{safe,k} \leftarrow \emptyset$
    for $\Gamma \in \Delta_{check}$ **do**
      if $K \models \neg(only_\Gamma \, \mathbf{U}^{=} \, H)$ **then**
        $\Delta_{safe,k} \leftarrow \Delta_{safe,k} \cup \{\Gamma\}$
      else
        $\Lambda_H \leftarrow \Lambda_H \cup \{\Gamma\}$
      end if
    end for
    $\Delta_{check} \leftarrow$ GeneratePowerSetLevel($F(K)$, Λ_H, $\Delta_{safe,k}$)
    $k \leftarrow k + 1$
  end while
  return Λ_H
end function

function GeneratePowerSetLevel(F, Λ_H, $\Delta_{safe,k}$)
  return $\{\Gamma \cup \{f\} \mid \Gamma \in \Delta_{safe,k} \wedge f \in F \setminus \Gamma \wedge \neg\exists\, \Gamma' \in \Lambda_H \,.\, \Gamma' \subseteq \Gamma \cup \{f\}\}$
end function

Algorithm 2. Overview of a DCCA algorithm for a hazard H and a fault-aware Kripke structure K. A maximum cardinality level k_{max} can be specified such that only minimal critical fault sets up to a maximum cardinality of k_{max} are computed. It is an anytime algorithm [30] that incrementally computes the minimal critical fault sets Λ_H for H by increasing cardinality. If the algorithm is aborted before it terminates by itself or $k_{max} < |F(K)|$, an incomplete set Λ_H might be returned; however, all fault sets contained in Λ_H are indeed minimal critical for H. It is therefore possible to abort long-running DCCAs and still profit from the incomplete results, as the most important minimal critical fault sets with the highest impact on hazard probability [80] are typically those of lower cardinalities that the algorithm finds first.
The algorithm conducts a bottom-up, breadth-first search through all fault sets $\Gamma \subseteq F(K)$, taking advantage of the monotonicity of the criticality property. Λ_H stores all minimal critical fault sets identified so far. The variable Δ_{check} contains the fault sets that have yet to be checked for a cardinality level k; the algorithm therefore begins by checking the empty set of faults for criticality. If a fault set $\Gamma \in \Delta_{check}$ is indeed critical, it is known to be minimal critical as otherwise one of its subsets would have to be critical as well, which would have been found in a previous iteration of the loop for some $k' < k$. Thus, Γ is added to Λ_H. If Γ is safe, by contrast, it is added to the loop-local variable $\Delta_{safe,k}$ that stores all safe fault sets of cardinality k.
Once all $\Gamma \in \Delta_{check}$ have been classified as either safe or critical, Δ_{check} is updated to contain all fault sets of cardinality $k+1$ that have to be checked next. To each safe fault set $\Gamma \in \Delta_{safe,k}$, the GeneratePowerSetLevel function adds another fault $f \in F(K) \setminus \Gamma$ such that $\Gamma' = \Gamma \cup \{f\}$ is not trivially critical, i.e., there is no subset $\Gamma'' \subseteq \Gamma'$ such that $\Gamma'' \in \Lambda_H$, and $|\Gamma'| = k + 1$. Subsequently, k is incremented and the newly computed fault sets $\Gamma \in \Delta_{check}$ of cardinality $k+1$ are checked for criticality.
The algorithm terminates and returns Λ_H at the latest when $k = |F(K)| + 1$, i.e., the set of all faults $F(K)$ is safe for H. More likely, however, either the maximum cardinality level k_{max} is reached or all minimal critical fault sets Λ_H are identified at some cardinality level $k < |F(K)|$; in the latter case, GeneratePowerSetLevel returns the empty set, subsequently causing the termination of the algorithm.

multiple initial states, the original DCCA would potentially classify some critical fault sets as safe in certain cases, which obviously is unacceptable. Overall, using safe fault sets to conduct DCCAs results in a couple of notable improvements in practice:

- **Initial States.** DCCA now supports multiple initial states without workarounds that construct a unique pseudo initial state by changing the model and the analyzed formula. Ironically, LTSmin only supports a unique initial state [116], forcing the S# framework to work around this limitation nevertheless.
- **Witnesses.** The model checker now generates a counterexample when a fault set Γ is critical, i.e., it constructs a witness that explains how Γ is able to cause the hazard. Hence, the model checking workflow is improved as originally, witnesses for safe fault sets showed how a hazard is not caused, which is typically of no practical use.
- **Tool Support.** The new variant of DCCA is compatible with a broader variety of modeling formalisms and model checkers as it can be formulated using either LTL or CTL. In order to check whether a fault set Γ is safe using CTL, Definition 5.15 could be reformulated to use the CTL formula $K \models \neg(only_\Gamma \, \mathbf{EU}^{=} H)$.
- **Persistency Constraints.** Similar to fairness conditions, persistency constraints cannot be enforced in CTL formulas [13]. Consequently, the use of LTL is required in order to specify persistency constraints when conducting DCCAs; with CTL, these constraints would have to be encoded into the analyzed models.

5.5.2 Completeness and Minimality

DCCA is a complete safety analysis technique [80] in the sense that the analyzed hazard cannot occur as long as at least one fault of each minimal critical fault set is never activated; thus, preventing the activations of all minimal critical fault sets effectively prevents the hazard from occurring. The minimal critical fault sets therefore show the weak points in the design of a safety-critical system, making it possible to devise additional safety measures to improve the system's overall safety. In the pressure tank case study, for example, the very first system design might not have included the timer, in which case activations of $\neg$is full alone could lead to tank ruptures. Safety analyses of this design revealed this single point of failure, so the timer was added to the controller in order to compensate for sensor failures. As a result, both $\neg$is full and $\neg$timeout are required to cause the hazard. DCCA's completeness formally backs this development approach for safety-critical systems, proving that it is indeed possible to circumvent hazard H by preventing the complete activations of all minimal critical fault sets $\Gamma \in \Lambda_H$.

Definition 5.18 (Completeness). *Let Δ be a set of fault sets causing some hazard H of a fault-aware Kripke structure K. Δ is* complete *if preventing the activations of at least one fault $f \in F(K)$ of each fault set $\Gamma \in \Delta$ prevents the occurrence of H, that is, $K \models (\bigwedge_{\Gamma \in \Delta} \neg \bigwedge_{f \in \Gamma} \mathbf{F}\, f) \rightarrow \mathbf{G} \neg H$.*

Theorem 5.3 (Completeness of DCCA). *The set of minimal critical fault sets Λ_H for some hazard H of a fault-aware Kripke structure K is complete.*

Proof. To construct a contradiction, let $\varsigma \in \mathit{paths}(K)$ be a path of K with $\varsigma \not\models (\bigwedge_{\Gamma \in \Lambda_H} \neg \bigwedge_{f \in \Gamma} \mathbf{F}\, f) \rightarrow \mathbf{G} \neg H$ or equivalently, $\varsigma \models (\bigwedge_{\Gamma \in \Lambda_H} \neg \bigwedge_{f \in \Gamma} \mathbf{F}\, f)$ and $\varsigma \models$

$\mathbf{F}\, H$. The second conjunct gives us some k with $\varsigma, k \models H$. Let $\Gamma' = \{f \in F(K) \mid \varsigma, i \models f \text{ for some } 0 \leq i \leq k\}$ denote the set of faults that are activated before or at step k. Consequently, Γ' is critical, i.e., $\varsigma \models \mathit{only}_{\Gamma'}\, \mathbf{U}^{=}\, H$, and, in particular, $\varsigma \models \bigwedge_{f \in \Gamma'} \mathbf{F}\, f$. Additionally, there is some minimal critical fault set $\Gamma'' \in \Lambda_H$ with $\Gamma'' \subseteq \Gamma'$. It follows that $\varsigma \models \bigwedge_{f \in \Gamma''} \mathbf{F}\, f$, which contradicts the first conjunct. □

There is the corner case of the empty fault set being minimal critical for hazard H, i.e., $\Lambda_H = \{\emptyset\}$. In that case, it is possible for the hazard to occur without any fault activations, indicating that the system has a safety-critical design flaw. Thus, DCCA fails to identify any causes as there are none, at least none related to the activations of one or more modeled faults. Consequently, the completeness theorem trivially holds as it is impossible to prevent the hazard by somehow suppressing a fault; the hazard can occur seemingly without a cause as it is not the result of any fault activations but rather results from a functionally incorrect system design. While Theorem 5.3 shows that all relevant causes are indeed identified by DCCA, it neither establishes traceability nor minimality on its own: There could still be many irrelevant fault sets $\Gamma \in \Lambda_H$ that do not actually contribute to the occurrence of H; the prevention of a fault $f \in \Gamma$ contained in such an irrelevant set does not necessarily affect whether the hazard occurs.

Definition 5.19 (Minimality)**.** *Let Δ be a complete set of fault sets causing some hazard H of a fault-aware Kripke structure K. Δ is* minimal *for H if for all $\Gamma \in \Delta$, $\Delta \setminus \{\Gamma\}$ is incomplete for H.*

Despite its high practical relevance, Theorem 5.3 is too weak to show the minimality of the set Λ_H computed by DCCA, necessitating a stronger claim:

Lemma 5.1. *Let Λ_H be the set of all minimal critical fault sets for some hazard H of a fault-aware Kripke structure K. All occurrences of H require the prior activation of all faults of at least one minimal critical fault set $\Gamma \in \Lambda_H$, that is, $K \models \mathbf{F}\, H \rightarrow \neg H\, \mathbf{U} \bigvee_{\Gamma \in \Lambda_H} \bigwedge_{f \in \Gamma} \mathbf{O}\, f$.*

Proof. For a contradiction, assume a path $\varsigma \in \mathit{paths}(K)$ with $\varsigma \models \mathbf{F}\, H$ such that $\varsigma \not\models \neg H\, \mathbf{U} \bigvee_{\Gamma \in \Lambda_H} \bigwedge_{f \in \Gamma} \mathbf{O}\, f$. Hence, $\varsigma, k \models H$ for some k but $\varsigma, k \not\models \bigvee_{\Gamma \in \Lambda_H} \bigwedge_{f \in \Gamma} \mathbf{O}\, f$ or equivalently, for all minimal critical fault sets $\Gamma \in \Lambda_H$ we have $\varsigma, k \models \bigvee_{f \in \Gamma} \neg\, \mathbf{O}\, f$. However, for $\Gamma' = \bigcup_{\Gamma'' \in \{\varsigma_F[i] \mid 0 \leq i \leq k\}} \Gamma''$, $\varsigma \models \mathit{only}_{\Gamma'}\, \mathbf{U}^{=}\, H$, thus Γ' is critical and there is a $\Gamma'' \subseteq \Gamma'$ that is minimal critical but $\Gamma'' \notin \Lambda_H$, a contradiction to the assumption that Λ_H contains all minimal critical fault sets. □

Theorem 5.4 (Minimality of DCCA)**.** *The set of minimal critical fault sets Λ_H for some hazard H of a fault-aware Kripke structure K is minimal.*

Proof. To construct a contradiction using Lemma 5.1, assume an irrelevant fault set $\Gamma^- \in \Lambda_H$ such that $K \models \mathbf{F}\, H \rightarrow \neg H\, \mathbf{U} \bigvee_{\Gamma \in \Lambda_H \setminus \{\Gamma^-\}} \bigwedge_{f \in \Gamma} \mathbf{O}\, f$. However, as Γ^- is minimal critical for H, there is some $\varsigma \in \mathit{paths}(K)$ such that $\varsigma \models \mathit{only}_{\Gamma^-}\, \mathbf{U}^{=}\, H$. Thus, there is a k such that $\varsigma, k \models H$ and for all $0 \leq i \leq k$, $\varsigma, i \models \bigwedge_{f \in F(K) \setminus \Gamma^-} \neg f$. Moreover, for all $\Gamma \in \Lambda_H \setminus \{\Gamma^-\}$, there is a $f \in \Gamma$ such that $f \notin \Gamma^-$; otherwise, $\Gamma^- \subseteq \Gamma$, a contradiction to the monotonicity of criticality as shown by Theorem 5.2 and the fact that Λ_H contains only minimal critical fault sets. Consequently, there is no $\Gamma \in \Lambda_H \setminus \{\Gamma^-\}$ such that $\varsigma, k \models \bigwedge_{f \in \Gamma} \mathbf{O}\, f$, that is, no minimal critical fault set is completely activated before H occurs in step k, a contradiction. □

DCCA always yields a complete and minimal set Λ_H of minimal critical fault sets, regardless of whether faults are injected into a fault-aware Kripke structure in accordance with Definition 5.7 or added in some other, arbitrary way. Adequacy, however, is only guaranteed for fault-aware Kripke structures $K_F \in K \triangleleft F$ with injected faults F. In this case, DCCA subsumes functional correctness with respect to a hazard H by checking the empty set of faults for criticality:

Proposition 5.7 (Subsumption of Functional Correctness)**.** *For a fault set F, fault-aware Kripke structures K and $K_F \in K \triangleleft F$, and a propositional logic formula H not referencing any faults $f \in F$, $K \models \mathbf{G}\,\neg H$ if and only if $\emptyset$ is safe for H in K_F.*

Proof. "⇒": $K \equiv_F K_F$ due to Proposition 5.1, thus $paths(K) \subseteq paths(K_F)$ and $paths(K) \neq \emptyset$ as Kripke structures are deadlock-free. For $\varsigma \in paths(K)$ with $\varsigma \in paths(K_F)$, we trivially obtain $\varsigma \models \neg(only_\emptyset\ \mathbf{U}^{=}\ H)$ as there is no k such that $\varsigma, k \models H$. For all other paths $\varsigma \in paths(K_F) \setminus paths(K)$, either $\varsigma \models \mathbf{G}\,\neg H$ or there is a k such that $\varsigma, k \models H$ and there is a $0 \leq i \leq k$ with $\varsigma, i \models \neg only_\emptyset$. Thus, at least one fault $f \in F$ must be activated before H occurs, so $\emptyset$ is safe.

"⇐": Let $\varsigma \in paths(K_F)$ with $\varsigma \models \neg(only_\emptyset\ \mathbf{U}^{=}\ H)$. If there is a k such that $\varsigma, k \models H$, there is a $0 \leq i \leq k$ such that $\varsigma, i \models \neg only_\emptyset$, hence $\varsigma \notin paths(K)$ as $\mathcal{A}(\varsigma) \neq \emptyset$. Otherwise, $\varsigma \models \mathbf{G}\,\neg H$ and $\varsigma \in paths(K)$ if and only if $\mathcal{A}(\varsigma) = \emptyset$ and thus $K \models \mathbf{G}\,\neg H$ as $K \equiv_F K_F$ due to Proposition 5.1. □

5.5.3 Fault Removal Optimization

Instead of using a series of LTL formulas to check for safe fault sets within a model, the model could alternatively be changed to make the checks more efficient: Faults $F(K) \setminus \Gamma$ are not allowed to be activated during a check of Γ for criticality due to $only_\Gamma$, so they could just as well be removed from the model entirely, reducing the number of states and transitions. Thus, instead of checking multiple formulas on the same model, a simplified formula could be checked for multiple reduced models. The following formalizes a fault removal variant of DCCA based on this idea, which generalizes an ad hoc approach to conduct DCCAs within the Scade tool [76]. To do so, an alternative characterization of safe fault sets $\Gamma \subseteq F(K)$ based on a different fault-aware LTL formula is first shown to be equivalent to Definition 5.15 under the assumption that the complement fault set $F(K) \setminus \Gamma$ is activation-independent in the analyzed Kripke structure:

Proposition 5.8. *Let H be a propositional logic formula not referencing any faults $f \in F(K)$ and $\Gamma \subseteq F(K)$ be a fault set such that $F(K) \setminus \Gamma$ is activation-independent in K. Γ is safe for hazard H if and only if $K \models \mathbf{G}\ only_\Gamma \rightarrow \mathbf{G}\,\neg H$.*

Proof. "⇒": Let $\varsigma \in paths(K)$ such that $\varsigma \models \mathbf{G}\ only_\Gamma$. We have to show that $\varsigma \models \mathbf{G}\,\neg H$. For a contradiction, assume that i is the first step in which H occurs, i.e., $\varsigma, i \models H$. Either $\varsigma, j \models only_\Gamma$ for all $0 \leq j \leq i$, a contradiction to $K \models \neg(only_\Gamma\ \mathbf{U}^{=}\ H)$, or there is a $0 \leq j \leq i$ such that $\varsigma, j \not\models only_\Gamma$, contradicting the assumption that $\varsigma \models \mathbf{G}\ only_\Gamma$.

"⇐": Let $K \models \mathbf{G}\ only_\Gamma \rightarrow \mathbf{G}\,\neg H$. For a contradiction, assume that there is a $\varsigma \in paths(K)$ such that $\varsigma \models only_\Gamma\ \mathbf{U}^{=}\ H$. Thus, there is a $k \geq 0$ with $\varsigma, k \models H$ and for all $0 \leq i \leq k$, $\varsigma, i \models only_\Gamma$. Either $\varsigma \models \mathbf{G}\ only_\Gamma$, a contradiction to $K \models \mathbf{G}\ only_\Gamma \rightarrow \mathbf{G}\,\neg H$. Alternatively, $\varsigma \not\models \mathbf{G}\ only_\Gamma$ and $l > k$ is the first step such that $\varsigma, l \not\models only_\Gamma$.

Due to activation independence of $F(K) \setminus \Gamma$, there is an infinite path fragment ς' for K with $\varsigma' \models \mathbf{G}\, only_\Gamma$ such that $\varsigma'' = \varsigma[l-1..]\varsigma' \in paths(K)$. Consequently, $\varsigma'' \models \mathbf{G}\, only_\Gamma$ and $\varsigma'', k \models H$, a contradiction to $K \models \mathbf{G}\, only_\Gamma \rightarrow \mathbf{G}\, \neg H$. □

Similar to how persistency constraints suppress undesirable fault activations, fault-aware LTL can thus be used to effectively suppress activations of faults $F(K) \setminus \Gamma$ during the check whether $\Gamma \subseteq F(K)$ is safe, at least as long as the fault set $F(K) \setminus \Gamma$ is activation-independent in K. It is then possible to remove these faults from the model instead of suppressing them with the formula as shown by Proposition 5.5, thereby replacing checks of multiple LTL formulas on the same fault-aware Kripke structure with all faults by a series of reachability checks on multiple reduced Kripke structures:

Theorem 5.5 (Fault Removal Optimization)**.** *Let H be a propositional logic formula not referencing any faults $f \in F(K)$ and $\Gamma \subseteq F(K)$ be a fault set such that $F(K) \setminus \Gamma$ is activation-independent in K. Γ is* safe *for hazard H if and only if $K_{\setminus(F(K)\setminus\Gamma)} \models \mathbf{G}\, \neg H$ for all $K_{\setminus(F(K)\setminus\Gamma)} \in K \setminus\!\setminus F(K) \setminus \Gamma$.*

Proof. Follows directly from Propositions 5.5 and 5.8. □

The proof of Theorem 5.5 generalizes and completes the one outlined for the ad hoc approach of conducting DCCAs in Scade [76]. In particular, at least one reduced Kripke structure $K_{\setminus(F(K)\setminus\Gamma)} \in K \setminus\!\setminus F(K) \setminus \Gamma$ is guaranteed to exist due to the activation independence in K that is assumed for $F(K) \setminus \Gamma$ as well as Theorem 5.1. Additionally, Proposition 5.3 establishes the adequacy of the fault removal optimization for injected, activation-independent faults $\Gamma' \subseteq F(K)$, i.e., the criticality of $\Gamma = F(K) \setminus \Gamma'$ can be determined by either removing Γ' or by injecting only Γ in the first place. It is thus not even necessary to construct the complete fault-aware Kripke structure containing all analyzed faults and to subsequently remove the faults that the analyzed DCCA formula would suppress anyway; instead, only the analyzed faults can be injected into the model, avoiding any potential analysis tool overhead when carrying out the fault removals.

By removing all faults from the analyzed Kripke structure whose activations would be discarded by the unoptimized DCCA formula, the fault removal optimization reduces the number of reachable states and transitions significantly: For a fault set $\Gamma \subseteq F(K)$, the number of transitions decreases by a factor of $2^{|F(K)|-|\Gamma|}$ in the best case. As minimal critical fault sets are usually rather small, these reductions can be significant and they add up, as an exponential number of checks has to be carried out in the worst case. Moreover, it is possible to replace the reachability formula $\mathbf{G}\, \neg H$ by an equivalent invariant check for $\neg H$, which avoids LTL model checking overhead [13, 116]. The overall potential analysis time reductions depend on the model adaptation overhead, the number of faults $|F(K)|$ to be analyzed, the size of the minimal critical fault sets, and potentially on some other tool-specific factors.

Table 5.2 shows a S#-based evaluation of the optimization's effects on analysis efficiency with notable analysis time reductions for some case studies as well as increases for others. S# provides two modes to conduct DCCAs: The unoptimized approach first pre-generates the state space of the analyzed system including all modeled faults before it subsequently checks all fault sets $\Gamma \subseteq F(K)$ for criticality; instead of LTL model checking, however, S# uses a slightly more efficient approach that integrates the DCCA

Case Study	# Faults	# Checked Fault Sets	# Minimal Critical Sets	Unoptimized Time	Optimized Time	
Height Control	13					
Collisions		53	4 (2.0)	83.6s	3.4s	24.6x
False Alarms		43	5 (1.0)	101s	2.9s	35.8x
Railroad Crossing	7					
Potential Collisions		11	6 (1.3)	3.4s	1.2s	2.8x
Hemodialysis Machine	9					
Contamination		54	3 (2.0)	43.8s	6.5s	6.7x
Dialysis Failure		14	6 (1.0)	35.9s	0.9s	39.8x
Personalized Medicine						
No Configuration	14	144	20 (1.8)	1.4s	1.7s	0.8x
No Configuration	18	1340	57 (2.0)	15.5s	13.1s	1.2x
No Configuration	18	3702	27 (2.0)	10.3s	38.0s	0.3x
No Configuration	23	76639	130 (3.3)	14.8m	65.9m	0.2x

Table 5.2. The results of the S#-based evaluation of the explicit-state safety analysis efficiency improvements, comparing both optimized and unoptimized DCCA times. Four S# case studies were evaluated on a 3.4 GHz quad-core CPU: The analyzed hazards are tunnel collisions and false alarms for the height control; trains on unsecured crossings for the railroad crossing case study; contaminated blood entering the patient's vein as well as overall dialysis failures for the hemodialysis machine; and situations in which reconfigurations of the personalized medicine production cell become impossible once all redundancy is used up. The table reports the number of faults contained in the model, the number of fault sets that had to be checked taking advantage of the monotonicity of the criticality property, the number of minimal critical sets and their average cardinality, the time with and without the optimization, as well as the overall analysis speedup or reduction. With and without the fault removal optimization, the same minimal critical sets are found and the same number of fault sets has to be checked for criticality.

formula directly into the model checker. Alternatively, the fault removal optimization can be enabled to check the criticality of $\Gamma \subseteq F(K)$ on-the-fly [124] with the other faults $F(K) \setminus \Gamma$ removed. In both cases, minimal critical fault sets are searched for using the bottom-up strategy. The height control, railroad crossing, and hemodialysis machine case studies show notable analysis time reductions when the fault removal optimization is enabled, at least halving analysis times.

The personalized medicine case study, by contrast, does not profit at all from the optimization; in fact, analysis times are significantly worsened it, taking up to 4.5 times longer than with unoptimized DCCA. There are two main reasons for this effect: The number of fault sets that have to be checked for criticality can become huge but execution of the case study model is extremely inefficient, computing only 215 transitions per second due to the case study's extremely large state vector. When the fault removal optimization is enabled, the S# model has to be executed up to 76639 times whereas without the optimization, it only has to be executed once to pre-generate the state space that is subsequently very cheap to analyze. For the largest configuration of the personalized medicine case study listed in Table 5.2, for example, state space generation takes the majority of the time with 844 seconds, while the subsequent 76639 checks for criticality are carried out much faster in roughly 42 seconds. For both DCCA variants, however, the personalized medicine case study clearly shows a scalability issue as the

number of fault sets that have to be checked increases exponentially for larger models. Chapter 8 discusses approaches for coping with this issue, bringing analysis times down to merely 4.6s even for the largest of the configurations shown in Table 5.2. With these additional optimizations, it becomes possible to analyze even larger configurations for which the full state spaces cannot be pre-generated due to memory constraints; for these configurations, the fault removal optimization is the only chance to conduct DCCAs.

5.6 Related Work

Limits of Model Checking. There are formal safety analysis techniques such as Formal Fault Tree Analysis that are based on interactive theorem proving instead of model checking [56, 80]. In contrast to model checkers, interactive theorem provers are only semi-automatic, requiring manual input and therefore making the verification more time-consuming. Model checking, on the other hand, is limited to systems with a finite amount of states, which makes model checking inappropriate for data-intensive systems [13]. In fact, many real-world systems quickly reach a size that precludes the use of model checkers due to computation time and memory constraints. One way to reduce the size of the state spaces is to use more aggressive abstractions, which, if not enough care is taken, can invalidate formal analysis results: The abstract models might no longer be adequate, that is, the actual real-world systems might expose nominal or off-nominal behavior that is not captured by the models. Therefore, abstractions are ideally carried out automatically by a tool in a way that the adequacy of the abstractions can be guaranteed. For example, counterexample-guided abstraction refinement is such a technique [43] that tries to automatically abstract from irrelevant states during analysis. Fault-aware modeling and fault removal DCCA can also be seen as automatic and adequate fault abstraction techniques.

Partial Order Reduction. Partial order reduction is an optimization technique that tries to reduce the amount of transitions that have to be considered during model checking [13, 164]. Instead of considering all transitions leaving a certain state, only a subset is considered, exploiting the fact that multiple sequences of transitions are often commutative, particularly when the high-level models of the analyzed system use asynchronous concurrency. Activation minimality of a fault-aware Kripke structure is similar to partial order reduction as it ignores irrelevant transitions "activating" faults without any effects. In a sense, activation minimality is partial order reduction for faults statically constructed into the model, therefore introducing less overhead during analyses. On the other hand, partial order reduction also optimizes transitions that are not related to faults, hence the technique could be combined with fault-aware modeling for further analysis efficiency improvements. In contrast to partial order reduction, activation minimality is not a true abstraction mechanism as it can affect whether a formula holds: For instance, an LTL formula $\mathbf{G}\,\neg f$ checking that $f \in F(K)$ is never activated might hold for some activation-minimal Kripke structure K, whereas the same formula likely does not hold when superfluous activations are allowed with state-based fault modeling. However, fault-aware modeling and specification openly changes the meaning of f within a formula from "f is currently in its active state" to "f was activated as it had an effect", while true abstraction techniques such as partial order reduction

or counterexample-guided abstraction refinement are internal optimizations that the analysis tools carry out automatically and transparently [43, 164].

Formal Fault Tree Analysis. In contrast to a non-formal fault tree, a formal one contains temporal logic formulas over the underlying system model in each event instead of a description in natural language. Additionally, all gates are given a precise temporal semantics that combines the input and output events appropriately [80, 157]. Each gate as well as its inputs and outputs can subsequently be analyzed in conjunction with the formal model, using either model checking or interactive theorem proving [56, 157, 158]. Similar to DCCA, Formal Fault Tree Analysis also determines fault sets that can cause the analyzed hazard, however, it cannot find only effective causes. For a minimal critical fault set $\Gamma \in \Lambda_H$ for hazard H, Formal Fault Tree Analysis in general either also finds Γ or it finds one of its subsets $\Gamma' \subseteq \Gamma$ [80]. Consequently, a completeness theorem similar to the one for DCCA given in Theorem 5.3 can also be proven for formal fault trees; the effectiveness of the identified fault sets, however, cannot be shown as the analysis technique cannot guarantee traceability [80]. Therefore, Formal Fault Tree Analysis in general considers a system to be less safe than it actually is.

Automatic Fault Tree Generation. Fault Tree Analysis is a technique that is often required by international norms in order to assess the safety of the system under development. During safety certification [186, 193], fault trees are particularly convenient: They document a decomposition of the system that clearly shows which faults cause errors and how these errors propagate through the system and eventually result in a hazard. Model checking-based analysis techniques such as DCCA, by contrast, do not provide such insights. It is trivially possible to construct flat fault trees from minimal critical fault sets; however, such fault trees are of no practical use as they do not show why and how the minimal critical fault sets were determined to cause the hazard [113]. For example, the FSAP tool and the Compass toolset [24, 151] generate such trivial fault trees based on their formal analyses. Other approaches use contract-based design to automatically generate hierarchically organized fault trees [29], requiring the manual specification of assumptions and guarantees for all system components. The resulting fault trees are structured along the component hierarchy, giving more insights into the internal error propagation chains and thus improving the fault tree's usefulness at the expense of an overall reduced level of analysis automation compared to DCCA.

Safety Analysis with BT Analyser. The symbolic LTL model checker BT Analyser [122] allows its search for counterexamples to be directed through the specification of constraints. To conduct safety analyses under the assumption that all modeled faults have permanent persistency, minimal critical fault sets can automatically be extracted from the counterexamples generated for the analyzed hazard, specified as an LTL formula. The constraints that are generated in order to guide the counterexample search take advantage of the monotonicity of criticality, overall resulting in an analysis technique that is very similar to DCCA. However, DCCA is more generally applicable: It does not restrict faults to permanent persistency, which is a strong assumption to make given that transient faults are the more general case, and it can be used with any model checker supporting either LTL or CTL.

NuSMV-Based Safety Analysis Techniques. Bozzano et al. [26] introduced a symbolic safety analysis technique integrated into the NuSMV model checker around the same time that Ortmeier et al. [159] developed DCCA. Even though the two techniques are similar and produce comparable minimal critical fault sets, they differ in their practical usability. The symbolic technique requires changes to the underlying model checker, generally allowing it to avoid the CTL or LTL model checking overhead introduced by DCCA; due to the optimized integration of DCCA into the S# framework, however, DCCA also profits from tight tool integration while still being tool-independent in general. Efficiency comparisons between DCCAs conducted by S# and safety analyses carried out by FSAP/NuSMV, xSAP, or the Compass toolset [24, 65, 151], all of which use the symbolic analysis technique, are still inherently unfair to either of the two sides: S# uses explicit-state model checking whereas NuSMV is a symbolic model checker. This difference predominantly influences analysis efficiency depending on the analyzed system and particularly its amount of nondeterminism, so the two techniques themselves cannot be easily benchmarked against each other.

Monotonicity. In contrast to DCCA, the symbolic techniques by Bozzano et al. [26] require all faults $f \in \Gamma$ of a fault set $\Gamma \subseteq F(K)$ to occur before a hazard in order for Γ to be considered critical. In general, the notion of criticality used by the technique is therefore not monotonic with respect to set inclusion, as additional fault activations can potentially make the larger set non-critical. It is thus necessary to consider all combinations of faults for criticality instead of following a bottom-up search strategy based on the level of cardinality similar to DCCA. Nevertheless, an optimization opportunity exists that allows NuSMV to prune paths during analysis that are known not to yield any minimal critical fault sets [26]. Under the assumption of monotonicity, the symbolic technique can be further optimized using SAT-based model checking [30], increasing the efficiency compared to the original analysis approach significantly: Bottom-up searches similar to DCCA become possible even though they are obviously incorrect if the assumption of monotonicity is violated. An additional search heuristic that only activates faults when they can indeed have an effect is conceptually similar to fault-aware modeling; however, no optimization similar to DCCA's fault removal can be taken advantage of as the underlying symbolic techniques must consider all modeled faults in unison. The SAT-based analysis approach also allows for underapproximations if analyses take too long as the technique guarantees the currently analyzed set of faults to be a subset of a minimal critical set. The analyses can thus be aborted after some time, potentially yielding some safe fault sets for which a minimal critical superset is known to exist. In such a situation, the system's safety is knowingly considered to be worse than it actually is, which can be an acceptable tradeoff for larger minimal critical fault sets that only marginally influence the probability of a hazard.

Incremental Safety Analysis. Safety-critical systems are typically developed iteratively and incrementally [161, 186], requiring multiple safety analyses at different stages of development as the minimal critical fault sets are likely to change over time. It can be useful to keep an eye on these changes as large differences might hint at modeling or specification errors [138]. In particular, late changes to a system design should only remove minimal critical fault sets or enlarge them, but they should never reduce any

previously found minimal critical fault sets or introduce new ones. However, when new functionality is added incrementally to the system design, new critical fault sets are typically introduced. When the comparison detects new minimal critical fault sets, it might have identified new behavior that has to be considered and approved under the point of view of overall system safety. Comparisons of minimal critical fault sets of different versions of the same product are also considered in Chapters 6 and 8: Previously determined safe or minimal critical fault sets can be used as inputs for future analyses, likely speeding them up as fewer checks have to be made to find all new minimal critical fault sets given the ones that are already known and are likely still valid.

Error Detection & Identification. While DCCA determines the cause consequence relationships between fault sets and hazards, it is only indirectly able to show the effectiveness of error detection and identification mechanisms typically built into safety-critical systems. Such mechanisms are supposed to monitor a plant, raising an alarm whenever a fault is detected by observing one of its effects [25]. Bozzano et al. [28] introduced a pattern-based language using epistemic temporal logic to formalize error detection and identification requirements based on the relation between observable signals and desired alarms. In particular, their formal framework allows to show the soundness and completeness of alarms, i.e., it is possible to determine whether the alarms are only raised when the faults were indeed activated and whether the absence of any alarms guarantees that the faults were not activated, respectively. Additionally, it is possible to characterize the diagnosability of the plant as well as the maximality of the diagnoser, showing that the available sensors are sufficient to diagnose the problem in the first place and that alarms are raised as soon as possible. DCCA can be used for similar analyses using violations of the aforementioned properties as hazards, potentially requiring some changes to the models, however, in order to be able to express the required formal properties.

Additional DCCA Variants. There are two additional variants of the original version of DCCA [159]: Adaptive-DCCA and Deductive Failure Order Analysis [75, 78]. The first variant replaces the specification of the hazard as a propositional formula with a more complex temporal one in order to support the analysis of self-organizing systems, allowing the hazard to occur multiple times before it is actually considered to occur. As discussed in Chapter 8, such an extension of DCCA can be avoided by following a systematic decomposition approach for the analysis of self-organizing systems. The second DCCA variant does not only determine whether a fault set $\Gamma \subseteq F(K)$ is minimal critical, it additionally checks the relative order of activations of faults $f \in \Gamma$. For example, DCCA computes the minimal critical fault set $\{\neg\text{is full}, \neg\text{timeout}\}$ for the hazard of tank ruptures in the pressure tank case study, but the hazard can only occur if $\neg$is full is activated before $\neg$timeout is activated. Consequently, Deductive Failure Order Analysis would only consider the fault sequence $[\neg\text{is full}, \neg\text{timeout}]$ to be minimal critical, whereas $[\neg\text{timeout}, \neg\text{is full}]$ would be classified as safe; both the S# framework and the Compass toolset, for instance, can be configured to check for such ordering dependencies [151]. The relative order of fault activations often provides additional information that can be helpful in designing hazard prevention mechanisms, although safety analyses become more time-consuming as additional checks have to be made.

Causality Checking. Leitner-Fischer [128] presents an approach for model checking-based causality checking that yields all effective causes for a consequence. Based on an event order logic that allows to formally express the occurrences and order of events, causal relationships between events can be deduced automatically, also capturing ordering information similar to Deductive Failure Order Analysis. The approach also supports probabilistic analyses, working around the efficiency issues of the probabilistic model checkers by a combination of probabilistic and non-probabilistic checks. Even for small-sized case studies with around 1.2 million states, 7.5 million transitions, and only five faults, non-probabilistic causality analysis has relatively high execution times and memory requirements compared to DCCA. In contrast to DCCA, however, causality checking generates all witnesses for the criticality of a fault set, which can be helpful in order to understand all the situations in which a fault set leads to the analyzed hazard. On the other hand, for some analyzed case studies, thousands of witnesses were reportedly generated which does not seem useful in practice. Ideally, a middle ground could be found between the two extremes of DCCA and causality checking that either generate one or all witnesses, respectively.

Summary and Outlook. Fault-aware modeling and specification focuses on fault activations, making them central to the Kripke structures by explicitly denoting the minimal set of faults that are activated by each transition. In contrast to the traditional state-based fault modeling approach, fault persistency is not encoded into fault-aware Kripke structures, instead relying on fault persistency constraints in the fault-aware LTL formulas analyzed over the Kripke structures. In general, fault awareness results in large reductions of the number of reachable states and transitions within a Kripke structure compared to state-based fault modeling: Even though analysis efficiency might in fact not improve in the worst case of degenerate models, realistic case studies such as the height control show efficiency improvements of up to three orders of magnitude. Moreover, the formal definitions of fault injection and fault removal clearly show the changes that can adequately be made to a model in order to incorporate or remove off-nominal behavior. After fault injection, DCCA can compute all minimal critical fault sets for a hazard using an anytime algorithm that checks fault sets by increasing cardinality due to the monotonicity of the criticality property, optionally taking advantage of the fault removal optimization to further speed up the analyses.

The high-level modeling languages that are used to create fault-aware Kripke structures are mostly irrelevant for the applicability of the formal techniques presented in this chapter as long as they are able to fulfill the prerequisites. For example, they must guarantee activation independence of the injected faults if the fault removal variant of DCCA is used. Moreover, they must adhere to the definition of fault injection in order to guarantee conservative extension for reasons of adequacy in accordance with the systematic fault modeling approach presented in Chapter 3. In particular, S# models can be analyzed with all of the formal techniques as explained in Chapter 6, taking advantage of fault awareness to reduce the impact of fault injection on analysis times as well as fault removal to speed up DCCAs. Furthermore, Chapter 6 also introduces heuristics for safe fault sets, allowing DCCA to combine its usual bottom-up search strategy with some top-down searches to decrease the number of criticality checks that have to be made. In Chapter 7, the formal techniques are used to analyze a S# model of the height control case study, showing their applicability in more detail. Furthermore, Chapter 8 extends the modeling and analysis approach to self-organizing systems, allowing the formal techniques presented in this chapter to be used at run time in order to cope with the unique challenges presented by self-organization.

Summary. S#'s integration of the explicit-state model checker LTSmin [116] allows for an analysis approach based on model execution that unifies non-exhaustive model simulation and fully exhaustive model checking. The unification guarantees semantic consistency between model simulations and formal analyses, including visual replays of model checking counterexamples as well as safety analyses based on DCCA. S# employs various optimizations to increase analysis efficiency, one of which relies on heuristics that try to guess safe fault sets of high cardinalities in order to reduce the number of fault sets that have to be checked for criticality.

Publications. The unified model execution approach is partially published in [82, 84]. DCCA heuristics are introduced in [120] while the analysis capabilities provided by the S# framework are described in the S# Wiki [102].

6 Unified Analysis of Executable Models

While S# models have a significantly higher level of expressiveness than fault-aware Kripke structures and are hence preferable for modeling safety-critical systems, Kripke structures have a formal semantics and can thus be exhaustively analyzed using a model checker. In order to enable formal analyses in general and DCCAs in particular, S# models must therefore either be explicitly converted into fault-aware Kripke structures or at least their execution semantics must be made compatible with the path semantics of Kripke structures. As illustrated by Figure 6.1, the S# framework uses the latter approach by executing models in such a way that fault-aware Kripke structures are implicitly generated on-the-fly for model checking, even though they never really exist at all during analyses. Nevertheless, the relation between S# models and fault-aware Kripke structures is important to establish as it allows the use of the formal techniques presented in Chapter 5 such as fault injection, fault removal, and DCCA. Section 6.1 consequently defines the Kripke structure semantics of executable models.

Section 6.2 subsequently discusses the S# framework's unified model execution approach in more detail. S# takes advantage of the explicit-state model checker LTSmin [116] that provides a programmable interface for the integration of various modeling languages,

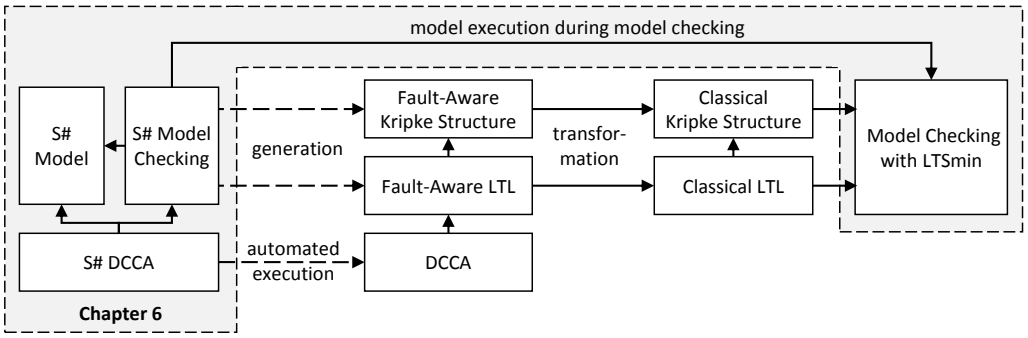


Figure 6.1. This chapter discusses a unified modeling and analysis approach for executable models. Conceptually, executable models integrate into the formal foundations established in Chapter 5 by generating fault-aware Kripke structures and fault-aware LTL formulas from the models. For reasons of efficiency, however, the model execution approach uses on-the-fly model checking based on LTSmin [116] to avoid the explicit creation of Kripke structures. As the model execution approach is nevertheless equivalent to fault-aware modeling and specification, it allows the use of the formal techniques introduced in Chapter 5 such as fault injection, fault removal, and DCCA, while also inheriting the efficiency gains over state-based fault modeling. Moreover, the S# framework can take advantage of the fault removal variant of DCCA.

allowing S# models to be executed using their regular C# and .NET semantics while they are being model checked. This model execution approach is unified with simulations and visualizations of S# models, guaranteeing that models are executed in the exact same way with consistent semantics regardless of whether a simulation is run or some formula is model checked. Unified model execution therefore forms the basis for all kinds of S#-based analyses as presented in Section 6.3, including simulations, visualizations, model checking, safety analyses using DCCA, model testing, and replays of model checking counterexamples. In order to further improve DCCA efficiency, S#'s DCCA implementation not only takes advantage of the fault removal optimization formally introduced in Chapter 5, but also allows the use of heuristics that try to guess safe fault sets of large cardinalities as explained in Section 6.4.

6.1 Kripke Structure Semantics of Executable Models

Figure 6.2 gives an overview of the structure of S# models as well as their partitioning into macro and micro steps. The fields of all component instances contained in a S# model constitute the model's state vector, i.e., the entirety of the information preserved between different macro steps to correctly describe the system's behavior. In the pressure tank case study, for example, the state vector consists of the pressure level within the tank, the time remaining until a timeout, a flag indicating whether the pump is active, and the software controller's state of its state machine. All of the variables in the state vector are initialized to some value; by default, C# zero-initializes all variables, e.g., integer fields are `0`, Boolean fields are `false`, and fields of reference types are `null`. The components can declare other initial values, and they can even choose them nondeterministically, in which case all possibilities are enumerated during model checking.

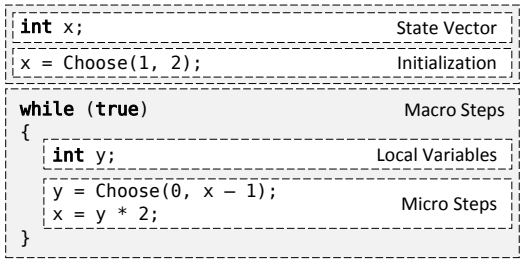


Figure 6.2. Conceptual overview of the structure of S# models: The state vector consists of globally declared state variables whose values are preserved across different macro steps; initial values of all state variables can be set nondeterministically. After initialization, an infinite sequence of macro steps is executed, taking all combinations of nondeterministic choices into consideration (not shown in the figure). In each macro step, step-local variables can be declared whose values are preserved within the same macro step but are lost as soon as a macro step ends. Micro steps consist of all C# statements executed during a macro step.

Once model initialization is complete, the actual execution of the model begins by computing an infinite sequence of macro steps. During each macro step, local variables can be used with C#'s regular stack-unwinding semantics; e.g., the value of a local variable declared in an `Update` method of some component is automatically discarded as soon as the method returns execution to its caller. All C# operations executed during a macro step are considered to be micro steps that are invisible to the model checker. It is the task of the S# framework to ensure that all nondeterministic micro steps involving one of S#'s `Choose` methods are executed exhaustively such that all combinations of nondeterministic choices are considered during model checking. Additionally, faults in S# also make use of the `Choose` methods to trigger their activations, hence all combinations of fault activations are inherently considered as they are simply mapped back onto S#'s standard nondeterminism mechanism.

Listing 6.1 shows a simple S# model whose state vector exclusively consists of the integer field `x` that can only have a value of zero, one, or two, with zero being its initial value. The instance of component `C` is the only component instance assumed to be contained in the model, hence each macro step corresponds to an invocation of `C`'s `Update` method. Thus, micro steps are executions of lines 8 to 11, 14 and 19. Activations of the fault `f` are also implicitly considered to be micro steps, that is, whenever the `GetValue` provided port is invoked, the S# framework checks whether the fault can be activated, which is always the case in this example, before it subsequently invokes the appropriate method override. As discussed in the next subsection, however, not all fault activations during the micro step execution phase are necessarily activation-minimal at the macro step level, requiring the S# framework to filter out spurious activations.

6.1.1 Formal Program Semantics

Kripke structure semantics cannot only be given to S# models, they can in fact be defined for all kinds of executable languages. For instance, the software model checker Java Pathfinder [194] analyzes programs written in the Java programming language [72] by

```
class C : Component
{
   int x = 0;
   Fault f = new TransientFault();

   public override void Update()
   {
      int y = 0;
      if (Choose(true, false))
         y = GetValue();
      x = x + y > 2 ? 2 : x + y;
   }

   public virtual int GetValue() => 1;

   [FaultEffect(Fault = nameof(f))]
   public class E : C
   {
      public override int GetValue() => 2;
   }
}
```

Listing 6.1. An exemplary S# model that is assumed to consist of a single instance of component `C`. The state vector only contains the integer field `x`, and there is a single transient fault `f`. `C`'s `Update` method is nondeterministic and indirectly affected by activations of fault `f` during invocations of the `GetValue` method. The local variable `y` is used to update the value of the state variable `x`.

```
y = 0
b = true or b = false

if b then
   if f then
      y = 2
   else
      y = 1
else
   skip

if x + y > 2 then
   x = 2
else
   x = x + y
```

Listing 6.2. The formal program that corresponds to the `Update` method of component `C` to the left; sequential composition is indicated by newlines to increase readability. The local variable b handles the nondeterministic condition of the original `if` statement. Invocations of the `GetValue` method and S#'s fault effect semantics are inlined. The value of the fault f can only be read in expressions whereas its actual value is determined by the Kripke structure semantics of executable models.

assigning Kripke structure-like semantics to Java byte code. The following therefore does not consider S# specifically, instead using a formally defined language with simple syntax and semantics for a more general representation of executable languages such as S# or Java. Additionally, the reduction to formal programs is made for reasons of brevity and comprehensibility, as high-level languages have many language features that are only important for conciseness and convenience but are otherwise irrelevant from a semantical point of view. The reduction comes without loss of generality, as formal programs are Turing complete and can therefore be converted to and from all other executable languages. In particular, S# inherits most of C#'s language features with the formal semantics given by Börger et al. [22] demonstrating that the complexity of the C#'s semantics would needlessly obfuscate the definition of the Kripke structure semantics for executable models.

The formal programs defined thereafter are illustrated in Listing 6.2 through the manual conversion of the S# model in Listing 6.1. The simple imperative language [150, 168] supports integer and Boolean variables as well as assignment, sequential composition, conditional, loop, and nondeterministic choice statements. No other types of variables, object-oriented concepts, or functions exist for reasons of conciseness; however, faults are explicitly represented in the language due to their importance for safety analysis and fault-aware Kripke structures that the formal programs are subsequently converted into. The formal programs support integer and Boolean literal values $v \in \mathit{Val}$ as well

as variables $x \in V$ of integer or Boolean types. Faults $f \in F$ are Boolean flags in this context, representing potential fault activations. During program execution, variable values are stored in variable environments $\sigma \in \Sigma \triangleq V \rightharpoonup \mathit{Val}$, i.e., partial functions that map each variable to the last assigned value. Thus, $\sigma(x)$ returns the value last stored in a variable $x \in V$, if any, while $\sigma[x \mapsto v]$ updates the value of x such that $\sigma[x \mapsto v](x) = v$ and $\sigma[x \mapsto v](x') = \sigma(x')$ if $x' \neq x$. The initial variable environment in which no values are stored yet is denoted by $\emptyset_\Sigma$.

Formal expressions are side effect-free and consist of literal values, read operations of variables, checks for potential fault activations, as well as a set of unary and binary operator applications uop and bop, respectively, that correspond to the usual operators supported by most executable languages:

$$e \in \mathit{Expr} \triangleq v \mid x \mid f \mid \mathit{uop}\ e \mid e_1\ \mathit{bop}\ e_2$$

The partial function $\mathcal{E}_\Gamma[\![-]\!] \colon \mathit{Expr} \to \Sigma \rightharpoonup \mathit{Val}$ defines the semantics of expressions relative to the potentially activated faults $\Gamma \subseteq F$. For a variable environment $\sigma \in \Sigma$, the given expression $e \in \mathit{Expr}$ is evaluated to its current value, provided that all variables referenced by e have previously been assigned. The semantics of unary and binary operators are defined as usual and are thus omitted.

$$\begin{aligned}
\mathcal{E}_\Gamma[\![v]\!]\sigma &= v \\
\mathcal{E}_\Gamma[\![x]\!]\sigma &= \sigma(x) \\
\mathcal{E}_\Gamma[\![f]\!]\sigma &= f \in \Gamma \\
\mathcal{E}_\Gamma[\![\mathit{uop}\ e]\!]\sigma &= [\![\mathit{uop}]\!](\mathcal{E}_\Gamma[\![e]\!]\sigma) \\
\mathcal{E}_\Gamma[\![e_1\ \mathit{bop}\ e_2]\!]\sigma &= [\![\mathit{bop}]\!](\mathcal{E}_\Gamma[\![e_1]\!]\sigma, \mathcal{E}_\Gamma[\![e_2]\!]\sigma)
\end{aligned}$$

The grammar for formal programs provides skip statements, variable assignments, sequential compositions, nondeterministic choices, conditionals, and loops. Potential fault activations cannot be influenced by any kind of statement, instead they are handled at the macro step level as discussed later on in order to adequately capture the assumption of zero execution time: Either a fault is active during all micro steps of a macro step, or it is dormant during all of them. If it is activated, all of the fault's effects that are somehow able to affect the system must be executed, that is, it is invalid to nondeterministically discard some of a fault's effects during the execution of a single macro step.

$$\rho \in \mathit{Prog} \triangleq \texttt{skip} \mid x \texttt{ = } e \mid \rho_1 \texttt{ ; } \rho_2 \mid \rho_1 \texttt{ or } \rho_2 \mid \texttt{if } e \texttt{ then } \rho_1 \texttt{ else } \rho_2 \mid \texttt{while } e \texttt{ do } \rho$$

The denotational semantics [150] of the language constructs given below highlights both the nondeterministic nature of the choice statement as well as the fact that program execution corresponds to macro steps that cannot be interrupted. The partial function $\mathcal{P}_\Gamma[\![-]\!] \colon \mathit{Prog} \rightharpoonup 2^{\Sigma \times \Sigma}$ defines the macro step semantics of formal programs with each statement of a program representing a micro step. The set of all fault-activated successors of a variable environment $\sigma \in \Sigma$ for a program $\rho \in \mathit{Prog}$ is denoted by $\mathcal{P}_\Gamma[\![\rho]\!]\sigma = \{(\Gamma, \sigma') \mid (\sigma, \sigma') \in \mathcal{P}_\Gamma[\![\rho]\!]\}$; the program $\rho \in \mathit{Prog}$ is terminating for variable environment $\sigma \in \Sigma$ if $\mathcal{P}_\Gamma[\![\rho]\!]\sigma \neq \emptyset$.

$$
\begin{aligned}
\mathcal{P}_\Gamma[\![\texttt{skip}]\!] &= \{(\sigma,\sigma) \mid \sigma \in \Sigma\} \\
\mathcal{P}_\Gamma[\![x \texttt{ = } e]\!] &= \{(\sigma,\sigma[x \mapsto \mathcal{E}_\Gamma[\![e]\!]\sigma]) \mid \sigma \in \Sigma\} \\
\mathcal{P}_\Gamma[\![\rho_1 \texttt{ ; } \rho_2]\!] &= \mathcal{P}_\Gamma[\![\rho_2]\!] \circ \mathcal{P}_\Gamma[\![\rho_1]\!] \\
\mathcal{P}_\Gamma[\![\rho_1 \texttt{ or } \rho_2]\!] &= \mathcal{P}_\Gamma[\![\rho_1]\!] \cup \mathcal{P}_\Gamma[\![\rho_2]\!] \\
\mathcal{P}_\Gamma[\![\texttt{if } e \texttt{ then } \rho_1 \texttt{ else } \rho_2]\!] &= \{(\sigma,\sigma') \in \mathcal{P}_\Gamma[\![\rho_1]\!] \mid \mathcal{E}_\Gamma[\![e]\!]\sigma = \text{true}\} \cup \\
&\qquad \{(\sigma,\sigma') \in \mathcal{P}_\Gamma[\![\rho_2]\!] \mid \mathcal{E}_\Gamma[\![e]\!]\sigma = \text{false}\} \\
\mathcal{P}_\Gamma[\![\texttt{while } e \texttt{ do } \rho]\!] &= \mu(\lambda W \,.\, \{(\sigma,\sigma') \in W \circ \mathcal{P}_\Gamma[\![\rho]\!] \mid \mathcal{E}_\Gamma[\![e]\!]\sigma = \text{true}\} \cup \\
&\qquad \{(\sigma,\sigma) \mid \mathcal{E}_\Gamma[\![e]\!]\sigma = \text{false}\})
\end{aligned}
$$

The skip statement has no effect and is simply discarded. Assignments update the variable environment accordingly, replacing the assigned variable's current value with the value of the evaluated expression. Sequential composition executes the two statements consecutively whereas choices nondeterministically execute one of their two statements. Conditional statements execute either of their two statements depending on whether the provided condition holds. Loops are executed as long as their condition continues to hold, requiring the computation of the least fix point [150] using the least fix point operator μ on the lambda function representing a single execution of the loop. If the number of loop iterations is infinite, model checking does not terminate. In such a situation, one of the model of computation's basic requirements is violated, namely that all macro steps are only allowed to contain a finite amount of micro steps.

6.1.2 Induced Fault-Aware Kripke Structures

Based on formal programs, executable models can be formalized as shown below; the actual connection to the S# framework's data structures is subsequently discussed in Section 6.2. The executability of the models is apparent in several ways compared to fault-aware Kripke structures: Instead of simple atomic propositions, executable models provide expressions over their state vector that can be evaluated to determine whether a proposition holds for some state. As for fault-aware Kripke structures, state expressions and faults are assumed to be orthogonal concepts, hence state expressions cannot reference any faults. The transition relation is implicitly specified by an execution program that allows the computation all successors for all states encountered during execution. Similarly, there is an initialization program that is executed to determine the initial values of all state variables contained in the model.

Definition 6.1 (Executable Models)**.** *An* executable model *is represented by the tuple* (P, F, S, V, E, I) *consisting of*

- *a set of state expressions* $P \subseteq Expr$ *not referencing any faults* $f \in F$*,*
- *a set of faults* F*,*
- *a set of state variables* S *comprising the state vector,*
- *a set of local variables* V *such that* $V \cap S = \emptyset$*,*
- *a terminating execution program* $E \in Prog$*, and*
- *a terminating initialization program* $I \in Prog$*.*

By convention, $P(M)$ *stands for* P*,* $F(M)$ *for* F*, and so on.*

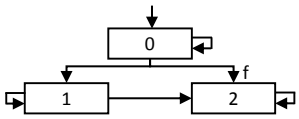


Figure 6.3. The fault-aware Kripke structure induced by the formal program in Listing 6.2. There are only three states in total with the label indicating the value of x; the value of the local variable y is irrelevant for the Kripke structure. Only one transition is able to activate fault f; all other potential activations are not activation-minimal at the macro step level.

The initialization and execution programs are required to be terminating as checking for termination is an undecidable problem. The S# framework, for instance, assumes termination, resulting in the overall model checking process of a S# model not terminating when either initialization or execution of the model is infinite. Additionally, formal programs do not support abnormal termination, for instance due to exceptions. The S# framework, by contrast, is in fact able to handle abnormal termination because of unhandled exceptions, performing the model checking process until the exception is encountered. Subsequently, the S# framework is able to generate a counterexample that shows how the exception can be generated, thereby providing additional information that helps to debug and fix such exceptions. Moreover, exceptions are occasionally useful for some types of analyses as discussed in Section 8.3, for example. Other reasons for abnormal termination, e.g., access violations due to unmanaged, out-of-bounds memory reads or writes, cannot be captured by the S# framework, thus resulting in an immediate termination of the entire model checking process.

Executable models induce fault-aware Kripke structures like the one shown in Figure 6.3 for Listing 6.2. Conceptually, such a fault-aware Kripke structure starts by executing the initialization statement $I(M)$ of the executable model M to compute all initial states $\sigma \in \Sigma$ for all combinations of activatable faults $\Gamma \subseteq F(M)$. Subsequently, each newly encountered state triggers a computation of the execution statement $E(M)$ for all activatable fault sets until no new states are found. During program execution, all possible combinations of potential fault activations are considered so that no activatable fault set is overlooked. However, some of these fault sets might only have an effect at the micro step level without any noticeably impact on the outcome of a macro step, i.e., they would result in transitions that are not activation-minimal. For example, the fault f of Listing 6.2 can be activated during micro step execution as it affects the value of the local variable y which is always zero-initialized at the point of the activation. At the macro step level, however, "activations" of f have no effect on the model's state unless x is zero, thus such spurious activations must be discarded during model checking as illustrated by the induced Kripke structure in Figure 6.3. Consequently, an activation minimization function *act-min* : $2^{2^F \times \Sigma} \rightarrow 2^{2^F \times \Sigma}$ is assumed such that $(\Gamma, \sigma) \in$ *act-min*(R) if and only if there is no $(\Gamma', \sigma) \in R$ with $\Gamma' \subsetneq \Gamma$, i.e., the function must filter out all non-activation-minimal transitions generated during program execution.

For each state encountered during model checking, the state expressions $P(M)$ are evaluated in order to determine the validity of the analyzed fault-aware LTL formula whose propositions $p \in \Phi$ refer to state expressions $p \in P(M)$. The local variables $V(M)$ are

not part of the induced Kripke structure's state as they represent intermediate results during executions of $I(M)$ and $E(M)$; such temporary computations are invisible for the model checker due to the assumption of zero execution time. In particular, the value of the local variable y of the formal program in Listing 6.2 does not have to be tracked in the state vector of the induced Kripke structure shown in Figure 6.3; within a single macro step, however, the micro steps of the original S# model shown in Listing 6.1 are executed using the regular C# and .NET semantics, hence y's value is preserved as long as the variable is in scope. As formal programs do not have a scoping concept, the values of all local variables must be cleared after macro step execution to avoid accidental reads of invalid values. Formally, the notation $\sigma' = \emptyset_{\Sigma}^{\sigma(S(M))}$ is used to denote the empty variable environment that only has the variables contained in the executable model's state vector $S(M)$ set to the values stored in the variable environment $\sigma \in \Sigma$. Thus, $\sigma'(x)$ does not exist if $x \notin S(M)$, whereas $\sigma'(x) = \sigma(x)$ for all $x \in S(M)$.

Definition 6.2 (Induced Fault-Aware Kripke Structures of Executable Models)**.** *The fault-aware Kripke structure* $K = (P, F, S, R, L, I)$ induced *by an executable model* M*, written as* $K(M)$*, is defined as follows:*

- $P = P(M)$,
- $F = F(M)$,
- $S = \Sigma$,
- $R = \{(\sigma, \Gamma, \emptyset_{\Sigma}^{\sigma'(S(M))}) \mid (\Gamma, \sigma') \in \textit{act-min}(\bigcup_{\Gamma \subseteq F} \mathcal{P}_{\Gamma}[\![E(M)]\!]\sigma)\}$,
- $L(\sigma) = \{p \in P \mid \mathcal{E}_{\emptyset}[\![p]\!]\sigma\}$*, and*
- $I = \textit{act-min}(\{(\Gamma, \emptyset_{\Sigma}^{\sigma(S(M))}) \mid (\Gamma, \sigma) \in \bigcup_{\Gamma \subseteq F} \mathcal{P}_{\Gamma}[\![I(M)]\!]\emptyset_{\Sigma}\})$.

Two conditions must be satisfied in order for an induced Kripke structure $K(M)$ to exist for an executable model M: Firstly, the initialization and execution programs must always assign a value to some variable $x \in V$ before x's value is read. Secondly, the state expressions cannot reference any local variables $x \in V(M)$ as the values of all local variables are unset in the states of the induced Kripke structure. The labeling function in Definition 6.2 explains why the state expressions $P(M)$ are not allowed to reference any faults $f \in F(M)$: State expressions are only evaluated over the empty set of faults for reasons of simplicity and efficiency. However, this is not a limitation as faults can indeed be referenced in fault-aware LTL formulas separately from propositions. All in all, these prerequisites guarantee the existence of an induced Kripke structure, allowing DCCAs to be conducted for executable models. In particular, executable models inherit the overall fault-aware modeling and specification approach introduced in Chapter 5, including the fault injection and fault removal operations.

6.2 Unified Model Execution

S# unifies LTSmin-based, fully exhaustive, explicit-state model checking and non-exhaustive simulations as illustrated by the architecture overview given in Figure 6.4. Both the simulator and the model checker provided by the S# framework execute S# models as regular .NET programs once they are compiled with the S# compiler. The S# compiler slightly extends the regular C# compiler in order to transform required ports, port bindings, and fault effects: Required ports are mapped to C# delegates that

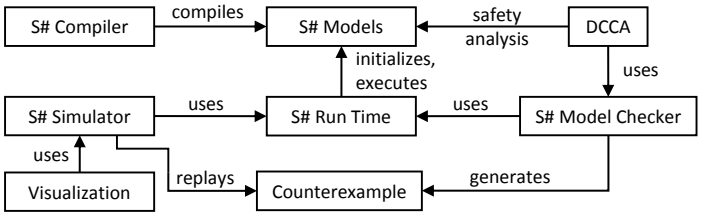


Figure 6.4. Illustration of the individual parts constituting the S# framework: The run time initializes S# models compiled by a slightly extended version of the C# compiler. Both the simulator and the model checker use the run time to execute a model. Model visualizations build upon the simulator, visualizing the simulated model state. Counterexamples generated by the model checker can be replayed by the simulator for debugging purposes. DCCA uses the model checker to check the criticality of all combinations of faults contained in a S# model.

are conceptually similar to required ports albeit with more verbose syntax. S# models establish port bindings using C#'s `nameof` operator in order to work around several C# language limitations that otherwise would have resulted in overly verbose binding specifications. Fault effects depend on some helper methods provided by the S# run time in order to check for fault activations, that is, to determine whether one of the effects or the original behavior of the affected port should be invoked.

S#'s unified model execution approach guarantees consistency between model checking and simulation as both the model checker and the simulator use the S# run time to execute the models. The only difference between simulation and model checking is that the latter is exhaustive, checking all combinations of nondeterministic choices and fault activations for each macro step whereas the former considers a single combination only. S#'s DCCA implementation builds upon the model checking integration provided by the S# framework, but the core algorithm that conducts complete DCCAs is independent of the analysis technique that is used to check fault sets for criticality; consequently, the S# framework could also be used to run DCCAs for other modeling formalisms and analysis tools as long as an appropriate analysis backend is implemented. S#'s model checker generates a counterexample when a formula is violated that can subsequently be replayed by the simulator. Counterexample replays can be visualized if a visualization is provided for a model, allowing for visual inspections of the model states encountered during model checking. While visualizations support reasoning about formula violations at the macro step level, the Visual Studio debugger can be attached during a counterexample replay to debug the individual micro steps executed by the S# model.

6.2.1 Model Execution Architecture

The class diagram in Figure 6.5 gives an overview of the classes and methods that implement the model execution approach of the S# framework. The ExecutionModel class implements the model execution semantics: It conceptually generates induced fault-aware Kripke structures on-the-fly in accordance with Definition 6.2. An ExecutionModel instance is created from a ModelBase instance, i.e., a composition of S# Component instances as illustrated by Section 4.5. Consequently, an ExecutionModel instance consists of a set of Component instances which in turn each have a set of Field instances that

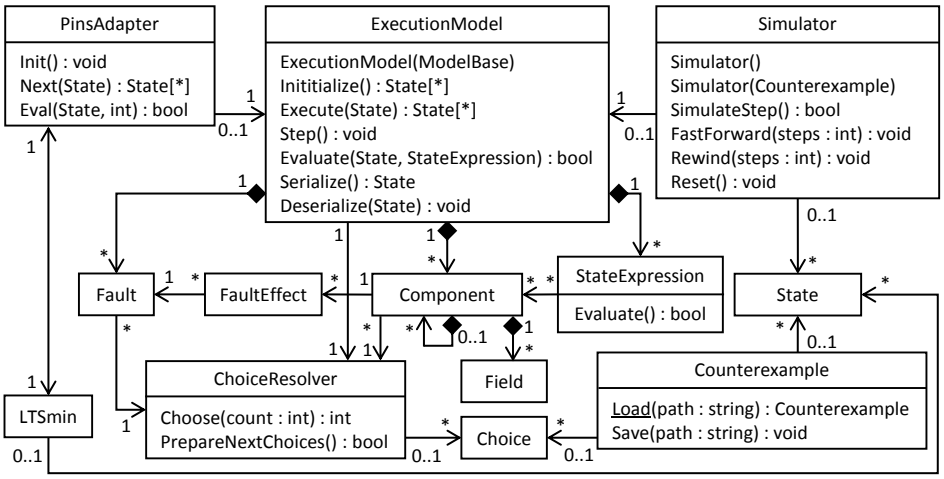


Figure 6.5. A UML class diagram showing the model execution-related parts of the S# framework, i.e., the classes and some of the methods required to simulate and model check S# models. The ExecutionModel class is responsible for evaluation of StateExpressions and model execution with the intended semantics; in particular, it uses a ChoiceResolver instance to determine all combinations of nondeterministic choices and fault activations. Thus, the Simulator class and LTSmin are decoupled from model execution semantics, only requiring them to manage States.

constitute the state of their associated Components. The Component instances correspond to instances of C# classes derived from the `Component` base class provided by the S# run time, whereas the Fields correspond to the class-scoped variables declared by a `Component`-derived class. The S# framework uses .NET's reflection capabilities [110] to search for the fields declared by a C# class. In the height control case study, for example, overheight vehicles are modeled as a Component with the Fields corresponding to a vehicle's position, its speed, as well as the lane it drives on. The combination of all Field instances transitively contained in an ExecutionModel makes up the ExecutionModel's state vector, different values of which are represented by instances of the State class.

Model State. For efficient storage and comparison, the S# framework uses fixed-size byte arrays as the run time representation of State instances. The method ExecutionModel::Serialize serializes Component states into a State instances, whereas ExecutionModel::Deserialize does exactly the opposite, deserializing a byte array back into Component and Field instances. During model checking, millions of states have to be stored, hence as few bytes as possible should be used to represent a single state. Additionally, many comparisons are carried out during model checking in order to determine whether a state has already been encountered before. These comparisons rely on state hashing and byte-by-byte equality checks and are also highly sensitive to the number of bytes required to store a State. For reasons of efficiency, S# therefore generates the two serialization methods dynamically at run time via reflection, tailoring them to a specific ExecutionModel instance to guarantee maximum efficiency with respect to serialization time and state storage size; Section 6.2.3 discusses some optimization techniques employed by the S# framework in order to reduce state vector sizes. Moreover, the use of fixed-size byte arrays for state storage explains S#'s limitations regarding object creation

Input: m : ExecutionModel, s : State
Output: a set of successor states $succs \subseteq$ State for s

$succs \leftarrow \emptyset$
while m.ChoiceResolver.PrepareNextChoices() **do**
  m.Deserialize(s)
  m.Step()
  $succs \leftarrow succs \cup \{\ m$.Serialize() $\}$
end while
return $succs$

Algorithm 3. Conceptual overview of the implementation of the ExecutionModel::Execute method. For all combinations of nondeterministic choices and fault activations determined by ChoiceResolver::PrepareNextChoices, the given state is deserialized using ExecutionModel::Deserialize. Subsequently, ExecutionModel::Step invokes the `Update` methods of all top-level Component instances contained in the model, thereby implementing the two-phase micro-macro step semantics of S# models, cf. Algorithm 1 on page 44. Once ExecutionModel::Step terminates, the ExecutionModel's state is serialized into a byte array using ExecutionModel::Serialize and stored in the successor set $succs$ that is eventually returned once the ChoiceResolver has exhaustively enumerated all nondeterminism contained in the ExecutionModel.

during model execution: Storing additional objects in the model's state vector would require dynamically-sized state arrays, making certain optimizations impossible or at least significantly more complex. On the other hand, new objects can be allocated during a macro step as long as no references to them are ever stored in the state vector.

Model Execution. The method ExecutionModel::Initialize generates all initial states of an ExecutionModel instance, resetting the Components' Fields to their original values by taking all nondeterministic initializations into account, if necessary. Subsequently, ExecutionModel::Execute is able to compute all successors of a state as conceptually illustrated by Algorithm 3. The ChoiceResolver::Choose method records the number of choices that can be made by a specific micro step during the execution of ExecutionModel::Step and returns the index of the chosen value; Fault instances internally use ChoiceResolver::Choose with some additional optimizations outlined in Section 6.2.2. Subsequently, the ChoiceResolver can ensure that all combinations of choices are exhaustively enumerated, returning the index of the next value to be chosen when the corresponding micro step is executed again. The ChoiceResolver is unaware of other sources of nondeterminism such as race conditions of threads, however, thus S# cannot guarantee exhaustive analyses in such cases, explaining its limitations around uncontrolled nondeterminism as discussed in Chapter 4. There are two reasons why a state might not have any successors at all or why there might not be any initial states: Either ExecutionModel::Initialize or ExecutionModel::Execute do not terminate or their execution is aborted abnormally due to an unhandled exception, for example. Both situations are readily discoverable during analyses as either model checking makes no progress or it is aborted prematurely.

Formula Evaluation. The StateExpression class represents arbitrary Boolean C# expressions that the S# framework evaluates over the Component instances in order to determine the validity of the analyzed LTL formula during model checking. The evaluation is expected to be terminating, deterministic, and side effect-free, otherwise the exact

behavior is unspecified. The method ExecutionModel::Evaluate evaluates a StateExpression instance for a given State, i.e., the State is deserialized using ExecutionModel::Deserialize, the corresponding C# code is executed and evaluated by invoking StateExpression::Evaluate, and the computed result is returned to the model checker that can subsequently check for violations of the overall LTL formula.

Executable Models. An ExecutionModel instance m formally defines an executable model M as follows: The propositions $P(M)$ are given by m.StateExpressions. The faults $F(M)$ simply correspond to the S# Fault instances contained in the model, i.e., m.Faults. The state variables $S(M)$ directly map to all Field instances contained in m. $V(M)$ is equivalent to all local variables declared by a C# program $\rho_{cs} \in Prog_{cs}$ with a function $\mathcal{L}\colon Prog_{cs} \to 2^V$ assumed to return all locally declared variables $x \in V$ of ρ_{cs}. The execution program $E(M)$ is m.Execute while the initialization program $I(M)$ corresponds to m.Initialize. If there are no modeling faults that cause the S# model m to get stuck in an infinite loop or to throw an unhandled exception, exploration of the induced Kripke structure terminates as soon as all reachable states are encountered. In this case, ExecutionModels always induce fault-aware Kripke structures without any deadlocks, so all paths through the Kripke structure are of infinite length and can thus be used to evaluate LTL formulas. In summary:

$$P(M) \cong \text{m.StateExpressions}, \qquad F(M) \cong \text{m.Faults},$$
$$S(M) \cong \bigcup_{\text{c}\,:\,\text{m.Components}} \text{c.Fields}, \qquad V(M) \cong \mathcal{L}(\text{m.Execute}) \cup \mathcal{L}(\text{m.Initialize}),$$
$$E(M) \cong \text{m.Execute}, \textit{ and} \qquad I(M) \cong \text{m.Initialize}.$$

In order to map m.Execute and m.Initialize to the world of formal programs, the induced Kripke structure can be considered to contain pointers to these two functions with $\mathcal{E}_\Gamma[\![-]\!]$ and $\mathcal{P}_\Gamma[\![-]\!]$ corresponding to regular .NET program execution.

Fault Injection & Removal. S# models trivially satisfy the formal definition of fault injection, i.e., Definition 5.7 on page 91: The faults and their effects introduce new states and make new state expressions possible with S#'s execution semantics guaranteeing that no fault effect is ever executed without a corresponding activation of the effect's associated fault. Consequently, new transitions and initial states are only generated in the induced Kripke structure precisely when the newly injected faults are activatable. Moreover, fault injection by S# cannot remove any behavior or state label, ensuring that all requirements of Definition 5.7 are satisfied. Therefore, S# models with injected faults are always a conservative extension of their original version without faults, both due to Proposition 5.1 on page 92 as well as due to S#'s fault execution semantics. On the other hand, the S# framework never removes injected faults from a model; instead, it makes use of the equivalent alternative justified by Figure 5.6 on page 97 of never injection faults that are later removed anyway. That is, fault removal is a zero-overhead operation in the S# framework as faults are removed from a model simply by never activating them during model execution. Activation independence of `TransientFault` and `PermanentFault` instances is trivially achieved as their attempted activations during model execution are unconditional.

LTSmin Integration. It is the responsibility of LTSmin to do the actual model checking: S# only executes an ExecutionModel, implicitly generating a Kripke structure that LTSmin

in turn checks [116] to determine whether an LTL formula is satisfied. However, neither the S# framework nor LTSmin ever explicitly build up a fault-aware or classical Kripke structure, instead using on-the-fly model checking algorithms [124]. If LTSmin detects a violation, it generates a Counterexample that consists of a sequence of State instances, which are trivial to deserialize back into sequences of ExecutionModel states using ExecutionModel::Deserialize. For later replays of micro steps, a Counterexample also captures the nondeterministic choices that the ChoiceResolver made during the generation of the Counterexample, which also includes information about fault activations. In order to enable the integration of various modeling languages, LTSmin provides the so-called PINS interface written in C [116] that S# makes use of. S#'s integration of LTSmin takes about 250 lines of C++/CLI code, a Microsoft-specific variant of C++ that allows for interoperability between C/C++ and C#. The PinsAdapter class maps LTSmin's C-based PINS interface to the C# interface of the ExecutionModel class: PinsAdapter::Init initializes and sets up an LTSmin instance, which in turn repeatedly calls PinsAdapter::Next to compute all successors of a serialized state using ExecutionModel::Execute. PinsAdapter::Eval prompts S# to evaluate a StateExpression instance identified by its index for some serialized state by calling ExecutionModel::Evaluate.

Simulation. Simulations of S# models work similar to model checking except that only a single path of the induced Kripke structure is explored. Simulator instances are either guided or unguided: Unguided simulations do not follow a predetermined path through the Kripke structure. Guided simulations replay the Counterexample instance passed via the constructor by forcing the nondeterministic Choices made by the model checker upon the Simulator. Consequently, Counterexample replays cannot only be stepped through state by state, but also allow debugging each micro step, giving insights into why and how some undesired state is reached. A simulation stores all computed states, allowing it to be fast forwarded or rewound by some number of steps using Simulator::FastForward and Simulator::Rewind, respectively. In contrast to Algorithm 3, Simulator::SimulateStep computes only one successor of the current state using the sequence of method calls ExecutionModel::Deserialize, ExecutionModel::Step, and ExecutionModel::Serialize based on a set of predetermined Choices; if a Counterexample is replayed, the method returns false once the last state of the Counterexample is reached. Simulator::Reset resets the simulated ExecutionModel to an initial state or the initial state of the replayed Counterexample.

6.2.2 Fault Execution

The generation of fault-aware Kripke structures from executable models seems to be counterproductive for the efficiency gains of fault-aware modeling and specification: After all, fault-aware Kripke structures K are designed such that the analysis tools do not have to consider all faults $f \in F(K)$ in every state, yet according to Definition 6.2, the S# framework has to do exactly that. That is, in order to determine all successor states of a state, the S# framework must compute at least $2^{|F(M)|}$ transitions plus any additional nondeterminism not related to fault activations. Even worse, the S# framework has some additional bookkeeping to do as it requires the *act-min* function to filter out superfluous transitions for fault "activations" that turn out to be non-activation-minimal at the macro step level. The only advantage that Definition 6.2 takes over from fault-aware

modeling is the reduction of the number of states as no information about active or dormant faults is tracked. Consequently, the direct implementation of Definition 6.2 in the S# framework would be detrimental to model checking performance; it is thus only used to provide the formal justification for the Kripke structure semantics of executable models, whereas the S# framework uses a significantly more efficient implementation that allows it to reduce the number of potential fault activations that it has to check. In the worst case, however, some S# models might indeed require the explicit enumeration of all combinations of potential fault activations, but all case studies analyzed with S# so far profit noticeably from S#'s optimizations.

The S# framework performs several automatic optimizations to reduce transition computation overhead. For example, fault activation minimization is performed on-the-fly instead of after-the-fact, reducing the number of minimality checks that have to be made. Moreover, fault removal is implemented by never activating the removed faults in the first place instead of injecting and subsequently removing them, hence fault removal is guaranteed to be faster than fault suppression via LTL. The three main optimizations carried out by the S# framework are explained in more detail in the following: The first two try to avoid the computation of non-activation-minimal transitions whenever possible by delaying fault activations and by skipping activations in situations where they are known to have no effect. The third one avoids the construction of fault persistency constraint formulas in fault-aware LTL, instead also taking advantage of model executability to cope with fault persistency.

Delaying Fault Activations. Definition 6.2 executes the initialization program once and the execution program repeatedly for all combinations of faults. Typically, most of the transitions generated in this way are not activation-minimal, therefore requiring some bookkeeping effort by the *act-min* function to remove all non-activation-minimal ones. The S# framework tries to avoid generating these superfluous transitions in the first place by delaying fault activations: Faults are only activated when the S# run time is absolutely sure that an activation must be attempted. Considering the S# model in Listing 6.1 again, for instance, there is no point in executing a macro step with an attempted activation of fault `f` if the `Choose` method in the `if` statement returns `false`, that is, the method `GetValue` is never invoked and thus `f` cannot have any effect. Consequently, the S# framework delays activations until the first time a fault effect associated with the fault is used in some way. This optimization usually reduces the number of transitions that have to be computed by an exponential factor. In the pressure tank case study, for instance, there are four faults, resulting in at least 2^4 transitions for each state of the induced Kripke structure without the optimization. The timer's ¬timeout fault and the pressure sensor's ¬is full fault, for instance, can only be activated while the tank is being filled, as the software controller only invokes the components' corresponding provided ports while the pump is running. The optimization therefore removes these two faults from consideration when the tank is not being filled, in these situations reducing the number of transitions by at least a factor of 2^2.

Skipping Fault Activations. To further increase analysis efficiency, the S# framework uses the fault activation process outlined in Figure 6.6 to skip an attempted fault activation in situations where the activation is known to be superfluous: If the fault effect's

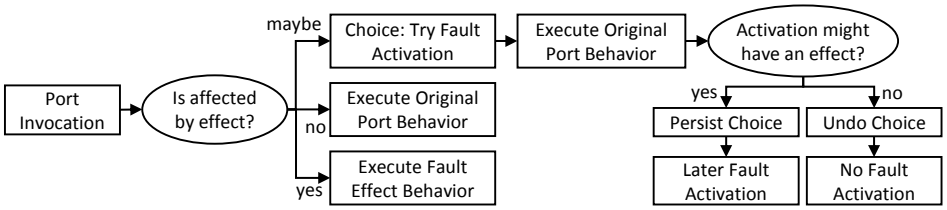


Figure 6.6. An overview of the fault activation process employed by the S# framework: Whenever a port is invoked that is potentially affected by a fault effect, a nondeterministic choice is made that conceptually either tries to activate the fault or that keeps it dormant. If the S# run time is able to determine with absolute certainty that the activation of the fault is not going to have an effect, i.e., it is irrelevant whether the overridden behavior of the fault effect or the port's original behavior is executed, the choice is undone. Otherwise the fault is marked for later activation, that is, the entire macro step has to be executed again, this time indeed activating the fault and executing the behavior of its effect.

behavior returns a constant value, a pattern that is often encountered in the case study models, the S# compiler generates code allowing the S# run time to check whether the value returned by the original behavior matches the value that the fault effect would return. If they match, the activation is known to be superfluous and an attempted activation would only result in a non-activation-minimal transition. In the pressure tank case study, for example, the software controller continuously checks for timeouts and sensor notifications about the tank being full while filling the tank, hence the fault delay optimization is ineffective. Skipping fault activations, on the other hand, avoids the computation of these non-activation-minimal transitions: The timer's `HasElapsed` and the sensor's `IsFull` provided ports both return `false` until the tank is almost fully filled up. As the effects of the ¬timeout and ¬is full faults simply return `false` in any case, the S# framework is often able to skip the faults' activations.

Evaluation of the Fault Activation Optimizations. The overall analysis efficiency improvements due to the fault activation optimizations are summarized in Table 6.1 for some of the case studies. For instance, only six non-activation-minimal transitions are computed for the pressure tank case study that are subsequently filtered out by the *act-min* function. The hemodialysis machine case study, by contrast, has more complex off-nominal behavior with side effects, making activation skipping ineffective: While fault activations are always delayed in any case, fault effects are allowed to have arbitrary side effects and thus activation skipping is disabled for effects whose behavior does not simply return a constant value. Even without both optimizations, all case studies profit from fault awareness as no fault persistency information is stored in the induced Kripke structures' states; for example, the height control case study is analyzed in roughly 12 minutes without fault activation-related optimizations instead of 1.5 days with state-based fault modeling, cf. Table 5.1 on page 106.

Fault Persistency Constraints. Before LTL formulas can be model checked, they have to be converted into Büchi automata [13]. During model checking, these automata are analyzed in conjunction with the actual system model, thus negatively affecting model checking efficiency if they introduce too many additional states. The worst

	Pressure Tank		Height Control		Railroad Crossing		Hemodialysis Machine	
Transition Count	747		57.4 mio		8.5 mio		9.8 mio	
Without Optimizations								
Computed Transitions	7,660	10.3x	7,171 mio	125x	224 mio	26.3x	185 mio	18.8x
Analysis Time	< 1s		12 min		21.4s		62.4s	
Delayed Activation								
Computed Transitions	1,857	2.5x	835 mio	14.5x	37 mio	4.4x	108 mio	10.9x
Analysis Time	< 1s		89.9s	8.1x	3.7s	5.8x	38.2s	1.6x
Both Optimizations								
Computed Transitions	753	1.0x	156 mio	2.7x	20 mio	2.5x	108 mio	10.9x
Analysis Time	< 1s		14.2s	50.7x	1.9s	11.3x	38.2s	1.6x

Table 6.1. Evaluation of the S# framework's automatic fault activation optimizations. The first line shows the number of activation-minimal transitions. The table reports the number of computed transitions and the overhead factor compared to the activation-minimal ones when disabling only the second or both optimizations; the first optimization cannot be disabled independently as the second one depends on the first one. The analysis times are reported for each optimization level, also showing the factor relative to the unoptimized analysis times. Without the optimizations, the number of computed transitions surpasses the number of activation-minimal ones by at least a factor of 10 for all case studies and even two orders of magnitude for the height control. By delaying fault activations, the number of computed transitions is reduced significantly and analysis times also decrease noticeably. With the second optimization, the pressure tank case study computes only six superfluous transitions, whereas there are no additional benefits for the hemodialysis machine: The fault effects of this particular case study are more complex than simply returning a constant value, effectively disabling the optimization. For the height control and the railroad crossing case studies, by contrast, the second optimization further reduces the number of transitions that have to be computed such that approximately only every third computed transition is not activation-minimal.

case complexity of Büchi automaton construction is exponential in the number of temporal operators, which is problematic as fault persistency constraints introduce many additional temporal operators. S# models, however, specify persistency constraints as `Fault`-derived classes that internally use regular C# code to determine fault activations. Therefore, faults in S# are conceptually already specified as Büchi automata, allowing the S# run time to avoid any automaton construction overhead. Furthermore, the `TransientFault` class provided by the S# run time is fully optimized such that transient faults do not introduce any overhead whatsoever, meaning that they only lead to new states and transitions in exactly those situations in which they are activatable. For permanent faults or other kinds of fault persistency, some state information must be tracked, but efficiency is still orders of magnitudes better than state-based fault modeling as seen in Table 5.1 on page 106 as well as Table 6.1.

6.2.3 State Storage and Serialization

State storage and serialization overhead has a significant impact on S#'s model checking times and memory consumption. For example, a state vector size of 100 bytes means that each state encountered during model checking consumes 100 bytes of memory

Offset	Length	Field	Comment
0	1	Pump::_isEnabled : bool	
1	7		padding
8	8	Tank::_pressureLevel : int	range-restricted
16	8	SoftwareController::_stateMachine::_state : State	enumeration
24	8	Timer::_remainingTime : int	range-restricted

Table 6.2. Overview of the pressure tank case study's state vector layout: The offset column shows each field's offset in bits from the start of the state vector, whereas the length column indicates the number of bits required to store the field denoted in the field column. Fields of non-Boolean types are grouped by the number of bytes required to store them, rounded up to the nearest power of two. Consequently, seven bits of padding must be added at offset 1 so that the 1-byte _pressureLevel field can be stored with the correct alignment at offset 8.

plus some overhead for look-up and storage by LTSmin; thus, roughly 86 million states can be stored when 8 GBytes of memory are available for state storage. Smaller state vector sizes not only increase the number of states that can be stored, they also decrease state management overhead: LTSmin uses hash tables [124] to efficiently determine whether a state returned by ExecutionModel::Execute has already been encountered before, i.e., states are hashed to 32-bit integers, the hash values are compared, and when the hash values are the same, the actual states are compared byte-by-byte to rule out hash collisions. State hashing and state comparison scales linearly with the state vector size, so smaller state vectors directly lead to lower state comparison overhead. Furthermore, the ExecutionModel::Serialize and ExecutionModel::Deserialize methods are called several times for each encountered state, so they also must be as fast as possible. The S# framework generates these methods dynamically using run time code generation to increase efficiency, but the number of fields that have to be read and written during state serialization and deserialization also impact their execution times. In order to reduce the overhead associated with state storage and state serialization, the S# framework automatically compacts state vectors to decrease their sizes. Sometimes, it is even possible to manually hide a field, removing it from the state vector entirely without affecting soundness and completeness of the analyses.

State Vector Compaction. The serialization methods generated by the S# run time group all fields by the number of bits required to store their values in order to reduce state vector sizes: Boolean values are stored as single bits, individual bytes as individual bytes, and so on. Additionally, range restrictions specified for a field are taken into account when doing these compactions, i.e., an integer field marked with `[Range(0, 5, OverflowBehavior.Error)]` is stored in a single byte; `enum`s are similarly compressed by considering the values of their literals. Object references are always stored as 2-byte values, thus allowing a model to contain at most 2^{16} different objects. Except for Boolean values, all values are stored in the lowest power-of-two byte count that is able to hold the value, grouped together by the number of bytes required to store a field. The starting address within the state vector of each such group is aligned to the same power of two to avoid potentially slow unaligned memory accesses. Table 6.2 shows the pressure tank case study's state vector layout using these state compaction techniques.

Field Hiding. Some fields store values that remain constant during model execution or that represent intermediate values that should not be part of a component's state. The former case often happens with references to subcomponents that are set during model initialization but are subsequently never changed during model execution. It thus makes no sense to continuously serialize and deserialize the component references, especially since reference serialization is particularly expensive for technical reasons. Moreover, there are situations in which some value must be preserved across different micro steps and accessible from different methods, necessitating the value to be stored in a field. If the field's value does not need to be preserved between different macro steps, however, the field should not be part of the state vector. To mark such fields, the S# framework provides the `[Hidden]` attribute, named after SPIN's `hidden` keyword that serves a similar purpose [98]. In the following example, `C`'s state only consists of the field `_i3`, as fields declared as `readonly` are implicitly considered to be `[Hidden]`:

```
class C : Component
{
   readonly int _i1;
   [Hidden] int _i2;
   int _i3;
}
```

The `readonly` keyword is a mechanism provided by C#, in this case ensuring that the value of `_i1` can only be set in a constructor of `C`. Consequently, the compiler can guarantee that the value stored in `_i1` indeed remains unchanged after the instantiation of a `C` instance. The `[Hidden]` attribute must be used in situations where C#'s restrictions are too limiting, e.g., a field might be initialized by a helper method that is invoked during model initialization. Alternatively, `_i2` might be updated during model execution, but its value might always be written before it is read in a macro step; hence, its value would not need to be preserved in the state vector. Chapter 7 uses the `[Hidden]` attribute for the vehicle's speed in the height control case study for this very reason, for instance. Hiding a field affects analysis soundness and completeness if the field should have been in the state vector, typically resulting in incomplete exploration of the induced fault-aware Kripke structure. At the time of writing, the S# framework provides no mechanism to check whether a field can adequately be hidden; for `readonly` fields, the C# compiler provides some guarantees that, however, can be undermined by the use of reflection which does indeed allow arbitrary changes to `readonly` fields.

Element Hiding. Fields of a type that implement `IEnumerable<T>` such as arrays or `List<T>`, among others, can be hidden just like all other fields. However, hiding the field does not implicitly hide its elements. A situation that is occasionally encountered is that references to subcomponents are stored in an array field. If neither the array nor the subcomponents change during model execution, both the array as well as the references to the subcomponents contained in the array can and should be removed from the state vector. For instance, Chapter 7 hides the vehicle instances stored in an array from the state vector of the height control case study model. The following component `C` declares two arrays containing references to subcomponents; `_a1` is completely hidden from the state vector, whereas for `_a2`, only the array reference stored in the field is hidden but the references contained in the array are not.

	Pressure Tank		Height Control		Railroad Crossing		Hemodialysis Machine	
With Compaction								
State Vector Size	4		12		12		40	
Time	< 1s		14.2s		1.9s		38.2s	
Without Compaction								
State Vector Size	24	6.0x	40	3.3x	40	3.3x	132	3.3x
Time	< 1s	1.0x	17.2s	1.2x	2.7s	1.4x	54.2s	1.4x

Table 6.3. An evaluation of the effects of state vector compaction on the overall analysis times as well as the number of bytes required to store a state. While the reduced state vector sizes do not have a large impact on analysis times, the improvements are nevertheless measurable. The main advantage of state compaction clearly lies in the reduced memory footprint, enabling analyses of models that are up to six times larger given a fixed memory budget.

```
class C : Component
{
   [Hidden(HideElements = true), Subcomponent]
   Component[] _a1;

   [Hidden(HideElements = false), Subcomponent] // or: [Hidden, Subcomponent]
   Component[] _a2;
}
```

Evaluation of the State Vector Optimizations. Table 6.3 compares state vector sizes and analysis times with and without state vector compaction, using field and element hiding in both cases. Table 6.4, on the other hand, contrasts compacted state vectors with and without field and element hiding. Both optimizations together noticeably reduce analysis time and memory requirements during model checking, albeit to a lesser extent than fault-aware modeling and specification or S#'s fault activation optimizations: The fault activation optimizations potentially remove exponential factors from the analysis, whereas the state vector optimizations often only affect constant factors. Field hiding, however, can also remove exponential factors when the values of the hidden fields are irrelevant but nondeterministic as for the speed of the vehicles in the height control case study. Overall, field and element hiding results in more notable analysis speedups in contrast to state compaction. The reason for the efficiency gains does not primarily lie in the smaller state vector sizes, but rather in the fact that most fields that can be hidden store constant references to subcomponents. Deserialization of such object references is expensive as it requires a dictionary look-up in the current implementation of the S# framework, explaining the negative impacts on analysis times.

6.3 Analyzing Executable Models

The S# framework provides three main analysis techniques based on its unified model execution architecture: LTL formulas can be exhaustively model checked, DCCAs can be carried out automatically, and non-exhaustive model simulations can be executed. DCCAs build upon S#'s integration of the explicit-state model checker LTSmin, whereas simulations are the basis for replays of model checking counterexamples, visualizations, and model tests. Moreover, all of these analysis techniques can be integrated into Visual

	Pressure Tank		Height Control		Railroad Crossing		Hemodialysis Machine	
With Field Hiding								
# States	680		1.3 mio		2.5 mio		0.4 mio	
# Transitions	747		57.4 mio		8.5 mio		9.8 mio	
State Vector Size	4		12		12		40	
Time	< 1s		14.2s		1.9s		38.2s	
Without Field Hiding								
# States	680	1.0x	1.8 mio	1.4x	2.5 mio	1.0x	0.4 mio	1.0x
# Transitions	747	1.0x	86.5 mio	1.5x	8.5 mio	1.0x	9.8 mio	1.0x
State Vector Size	12	4.0x	76	6.3x	32	2.7x	512	12.8x
Time	< 1s	1.0x	91.2s	6.4x	6.3s	3.3x	7.3m	11.5x

Table 6.4. An evaluation of the effects of field and element hiding on the sizes of the state vectors and analysis times: Without hiding, memory consumption increases between 2x and 12.8x with analysis times going up by similar factors. For the height control case study, state and transition counts increase as well because of the removal of the vehicle's speed field from the state vector. The hemodialysis machine case study also has hidden fields that do not remain constant without affecting the outcome of the analyses, but unhiding them causes such an enormous increase in additional states that they were kept hidden for the evaluation. For the other case studies, all hidden fields have constant values during model execution.

Studio; for instance, model checking, DCCA, and simulations can be carried out using testing frameworks such as NUnit [152], allowing Visual Studio's test runner to display the analysis results. The integration with standard testing tools for .NET also facilitates automated regression testing on build servers that uncover accidental changes to a model when some model checked formulas no longer hold, for example.

Just like S# models can be composed together using arbitrary C# code, cf. the model of the height control case study presented in Chapter 7, analyses can also be set up and executed using the full flexibility of C# and .NET: The S# run time provides programmatic control over model checking and simulations, in particular returning model checking counterexamples as sequences of actual model states that can be interpreted in various ways; for example, a counterexample can be output to the console, visually replayed using a simulation, or debugged by attaching Visual Studio's C# debugger to a simulation. Consequently, S#'s .NET heritage is taken advantage of in all phases of a model's life-cycle: During development, Visual Studio's code editing and refactoring features support the creation of the models, model instantiation can take advantage of flexible composition strategies based on various .NET technologies such as reflection, and model analysis can be tightly integrated into the entire Visual Studio development process and associated tooling. The S# model life-cycle is illustrated by Figure 6.7.

6.3.1 Model Checking Executable Models

The S# framework provides two model checking backends: The first one is based on an integration of the explicit-state model checker LTSmin [116] and the second one is a custom, C#-based implementation of a model checker based on LTSmin's sophisticated on-the-fly model checking algorithms [124]. The S# model checker was developed as the integration of LTSmin has some rough edges due to the different programming

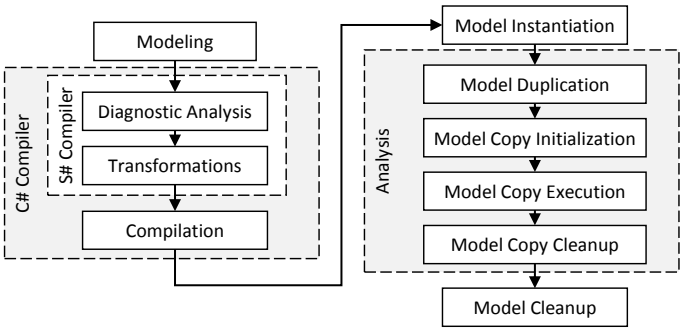


Figure 6.7. Overview of the model life-cycle in the S# framework starting with the creation of a S# model: The S# compiler is an extension library for the C# compiler that runs some S#-specific diagnostics over the model, ensuring, for example, that the ports bound together by an invocation of the `Bind` method actually exist. Subsequently, the S# compiler transforms required ports, fault effects, and port bindings such that the resulting code becomes executable C# code with the desired semantics. The transformed code is compiled to a regular .NET assembly by the standard C# compiler. For analyses, an instance of the model must be created. Such an instance is copied for each analysis, allowing the S# run time, for example, to use multiple independent threads for model checking. The model copies are initialized, executed for analysis, and cleaned up afterwards. The original model instance is cleaned up as soon as it is no longer needed for further analyses, causing .NET's garbage collector to free the model's allocated memory.

languages, platforms, and tools that LTSmin (C, GCC, Linux) and S# (C#, Visual Studio, Windows) are based on. As S#'s model checker is a .NET reimplementation of a subset of LTSmin, model checking with S# is generally considered to be LTSmin-based, regardless of which model checking backend is actually used. Table 6.5 gives an overview of the features supported by the two model checking backends; in general, the use of S#'s own model checker is preferred unless LTL model checking is required.

In the following example, an invariant of a S# model of the pressure tank case study is model checked. The model is instantiated in line 3 without any sophisticated instantiation techniques due to the case study's simplicity. Line 4 suppresses the activations of all faults contained in the model, i.e., the model is checked as if no faults had been modeled in the first place. The instance of S#'s model checker initialized in line 6 is used to carry out the analysis in line 7. For invariant checks, the `result` returned by the invocation of the `CheckInvariant` method indicates that the formula holds if it is satisfied by all reachable states. In this example, the invariant `!model.Tank.IsRuptured` is analyzed; it holds as the tank cannot rupture without any fault activations. Consequently, a success message is written to the console in line 10. Otherwise, the `result` contains a counterexample that can be immediately replayed or saved to disk for later replay as shown in line 12. It is also possible that some unhandled exception occurs during model checking; such exceptions are wrapped in an `AnalysisException` instance that can be caught as shown in line 14. In this case, an error message is written to the console in line 16 that includes the message of the original exception, and the counterexample generated for the exception is saved to disk. However, not all `AnalysisExceptions` contain a counterexample, hence the use of C#'s conditional access operator `?.` in line 17. For instance, no meaningful counterexamples can be generated for `OutOfMemoryExceptions`.

	S# LTSmin Integration	LTSmin-Inspired S# Model Checker
Multi-Core Support	×	✓
Invariant Checks	✓	✓
LTL Checks	✓	×
DCCA	×	✓
Hidden Fields in Formulas	×	✓
Counterexamples	×	✓
Exceptions	×	✓

Table 6.5. Comparison between the two model checking backends provided by the S# framework. As S#'s own model checker is inspired by the algorithms of LTSmin [124], the efficiency of both model checkers is roughly equivalent. However, there are some integration issues that do not reliably allow the use of LTSmin's multi-core algorithms, hence multi-core model checking with LTSmin is disabled. If it does work, LTSmin achieves the same speedup on a quad-core CPU as S#'s own model checker. The only disadvantage of the S# model checker is its lack of support for LTL model checking at the time of writing. However, the model checker is nevertheless used to conduct DCCAs: The fault removal variant only requires invariant checks and S# does not encode persistency constraints in LTL formulas. Moreover, only the S# model checker supports accesses to hidden fields in formulas and the generation of counterexamples. In particular, counterexample can be generated for unhandled exceptions thrown during model checking.

```
try
{
   var model = new Model();
   model.Faults.SuppressActivations();

   var checker = new SSharpChecker();
   var result = checker.CheckInvariant(model, !model.Tank.IsRuptured);

   if (result.FormulaHolds)
      Console.WriteLine($"Formula holds for all {result.StateCount} states.");
   else
      result.Counterexample.Save(/* file name */);
}
catch (AnalysisException e)
{
   Console.WriteLine($"Exception: {e.InnerException.Message}");
   e.Counterexample?.Save(/* file name */);
}
```

6.3.2 Deductive Cause Consequence Analysis

The S# framework carries out DCCAs completely automatically by collecting all faults F contained in a model and checking each combination of these faults for criticality. S# effectively carries out the fault removal variant of DCCA as it removes all other faults Γ' from the model when it checks a fault set $\Gamma = F \setminus \Gamma'$ for criticality, i.e., the activations of all faults $f \in \Gamma'$ are suppressed during model execution. Furthermore, the S# framework implements the DCCA algorithm illustrated by Algorithm 2 on page 112, thereby taking advantage of the monotonicity of the criticality property, i.e., all fault sets

are analyzed by increasing cardinality. For the pressure tank case study, for example, a DCCA is conducted for the model instance initialized below in line 1 by the invocation of the `SafetyAnalysis.AnalyzeHazard` method in line 2, passing along the model that should be analyzed as well as the hazard, i.e., a tank rupture in this case:

```
1 var model = new Model();
2 var result = SafetyAnalysis.AnalyzeHazard(model, model.Tank.TankRupture);
3
4 result.SaveCounterexamples(/* path */);
5 Console.WriteLine(result);
```

The returned `result` provides access to all minimal critical fault sets found by DCCA as well as the counterexamples generated for them. When DCCAs are carried out as regression tests in a continuous build environment, for example, it is thus possible to assert that the minimal critical fault sets match those that have previously been identified; otherwise, some potentially unintended change to the model invalidated previous safety analyses and might therefore require further consideration [138]. S#'s DCCA implementation is aware of unhandled exceptions, treating all fault sets for which unhandled exceptions are thrown as minimal critical; this feature is used by the testing approach for self-organization mechanisms presented in Section 8.3, for example. As shown by line 4 above, the `result` object provides a helper method `SaveCounterexamples` that writes the counterexamples for all minimal critical fault sets to the provided path for later replay. Furthermore, the `result` can be written to the console as shown in line 5, printing the following summary for the pressure tank case study:

```
1 Elapsed Time: 00:00:00.5703065
2 Fault Count: 4
3 Faults: SuppressIsEmpty, SuppressIsFull, SuppressPumping, SuppressTimeout
4
5 Checked Fault Sets: 13 (81% of all fault sets)
6 Minimal Critical Sets: 1
7
8  (1) { SuppressIsFull, SuppressTimeout }
```

The summary shows the time required to conduct the complete analysis in line 1. Due to the simplicity of the case study, the entire DCCA takes only around half a second before the single minimal critical fault set $\{\neg\text{is full}, \neg\text{timeout}\}$ reported in lines 6 and 8 is discovered. Moreover, the summary shows the number and names of the faults that are analyzed in lines 2 and 3 as well as the number of fault sets that have to be checked for criticality in line 5. For the pressure tank case study, 81% of all fault sets have to be checked for criticality, so the monotonicity of the criticality property does not help much to reduce the number of required checks in this case.

6.3.3 Simulating, Testing, and Visualizing Executable Models

Model simulations form the basis for model tests, model visualizations, and replays of model checking counterexamples. For example, model tests enable regression testing on a build server whenever changes are made to a model and visualizations allow for visual investigations of the effects of fault activations on the global system behavior. Counterexamples are often most intuitive to understand visually [61] instead of the textual outputs often provided by model checkers such as SPIN or NuSMV [38, 98].

Simulation. The following example shows the typical usage of the `Simulator` class and related helper methods provided by the S# framework. In line 1, a new instance of the pressure tank case study model is created; the activations of all faults contained in the model are suppressed in line 2 with line 3 enforcing the activation of the pressure sensor's ¬is full fault, that is, whenever ¬is full can be activated, it must be activated. Nondeterministic activation is not possible in a simulation as only one path through the induced fault-aware Kripke structure is considered, i.e., activations of each fault must either be suppressed or enforced. A new `Simulator` instance is created for the model in line 5. It is possible to simulate a single step as in line 6 or to advance the simulation multiple steps as shown in line 7. Furthermore, some of the previously simulated steps can be undone or the simulation can be reset to the model's initial state as illustrated by lines 8 and 9, respectively. Fault activations can always be enforced or suppressed between different invocations of any `Simulator` methods.

```
1 var model = new Model();
2 model.Faults.SuppressActivations();
3 model.Controller.Sensor.SuppressIsFull.ForceActivation();
4
5 var simulator = new Simulator(model);
6 simulator.SimulateStep();
7 simulator.FastForward(steps: 120);
8 simulator.Rewind(steps: 10);
9 simulator.Reset();
```

Model Testing. Unit, integration, or system tests can be written for S# models or individual components contained within them using regular .NET testing frameworks such as NUnit [152]. In the early stages of development, for example, new design ideas can quickly be tested and tried out, even allowing for an exploratory, test-driven development of the safety-critical system. Tests of individual components, groups of components, or the entire model serve as regression tests whenever changes to the model or to the design of the system become necessary. Tests that check the system's modeled effects of fault activations or tests that result in the occurrence of a hazard can also be used as templates for possibly manual hardware-in-the-loop tests or simulations of the actual system during later stages of the development process [186].

The following example shows an integration test of the pressure tank case study based on the NUnit test library: It asserts that there is a tank rupture precisely after the first 62 simulated macro steps when both the activations of ¬is full and ¬timeout are enforced. The test method declared in line 2 is marked with the `[Test]` attribute so that it can be executed with Visual Studio's integrated test runner. The model is initialized in line 4 and all of its faults except for ¬is full and ¬timeout are suppressed in lines 5 to 7. The `Simulator` instance is initialized in line 9, subsequently the simulated copy of the model is retrieved from the `Simulator` in line 10. As indicated by Figure 6.7, all analysis APIs provided by the S# framework operate on copies of the original model instance in order to support multi-threaded analyses. The original `Model` instance created in line 4 is therefore not affected by the simulation. The `for` loop declared on line 12 advances the simulation by a total of 61 steps in line 14. Line 15 uses the `Assert` class provided by NUnit to ensure that after each step, the tank has not yet ruptured. In line 18, the 62nd step is simulated which should result in a tank rupture as asserted by line 19.

```
[Test]
public void TankRupturesWhenSensorDoesNotReportTankFullAndTimerDoesNotTimeout()
{
   var model = new Model();
   model.Faults.SuppressActivations();
   model.Controller.Sensor.SuppressIsFull.ForceActivation();
   model.Controller.Timer.SuppressTimeout.ForceActivation();

   var simulator = new Simulator(model);
   model = (Model)simulator.Model;

   for (var i = 0; i < 61; ++i)
   {
      simulator.SimulateStep();
      Assert.IsFalse(model.Tank.IsRuptured);
   }

   simulator.SimulateStep();
   Assert.IsTrue(model.Tank.IsRuptured);
}
```

Test-Driving Actual System Components. While the test above only considers components modeled with S#, actual hardware components such as electric or hydraulic subsystems or real controller software can also be integrated into S# models to allow for hardware- or software-in-the-loop testing [12, 51, 52]. However, tests incorporating real hardware usually take more time as they must be executed taking the real-time constraints of the hardware into account, whereas the model-based tests execute much more quickly, only depending on how long it takes the S# framework to simulate the requested number of steps. Additionally, hardware-in-the-loop tests for hazards such as the tank rupture test above can be problematic, as in this case, the test would actually damage the hardware component under test, that is, the test must be aborted shortly before the hazard actually occurs [186].

The following example shows a simplified excerpt of the implementation of the pressure tank controller in the C programming language for an Arduino microcontroller [14]. There are two pins declared on line 1, one that is used to read the output of the sensor in line 4 and one that is used to enable or disable the pump in line 6. The controller software computes the `enablePump` value that is used in line 6 to decide whether the pump should remain active; the actual controller logic is omitted as it is conceptually similar to the state machine shown in Figure 3.17 on page 46. The `loop` function declared on line 2 is invoked once during each macro step in order to check the sensor value and to update the pump's activation state, thus representing invocation's of the software controller's `Update` method in the S# model.

```
int isFullPin, pumpPin;
void loop()
{
   int isFull = digitalRead(isFullPin);
   // Controller logic to compute enablePump omitted
   digitalWrite(pumpPin, enablePump ? HIGH : LOW);
}
```

Using S#'s support for variability modeling and flexible model composition, real hardware and software components can be integrated into S# models as shown in Figure 6.8 to allow for simulations or potentially even model checking. There are two ways to

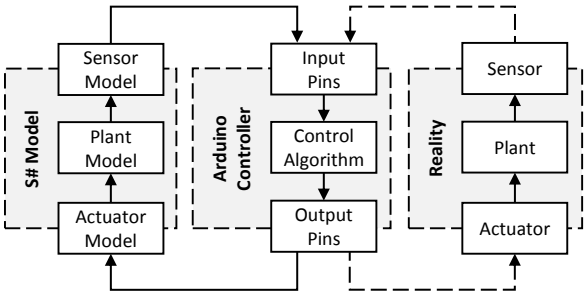
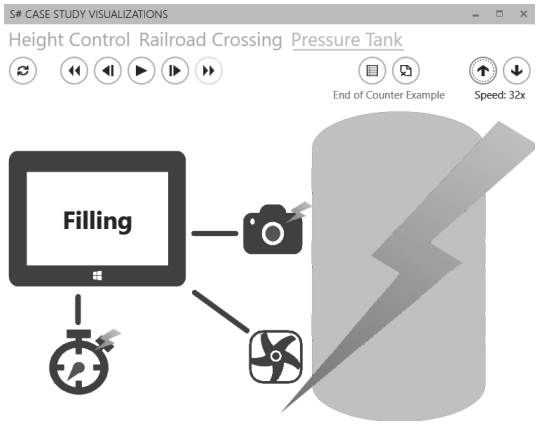


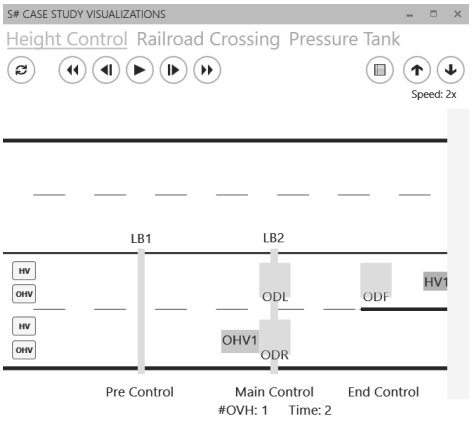
Figure 6.8. Conceptual overview of the integration of actual control software implemented for an Arduino microcontroller into a S# model: The S# model contains models of the actual sensors, plants, and actuators. The modeled sensors and actuators are mapped to the input and output pins of the control software, respectively, instead of connecting the pins to the real sensors and actuators. Consequently, the effects of sensor and actuator faults on the actual controller can be analyzed using the S# framework. With standard object-oriented programming techniques, the interface between the S# model and the actual control software can be generalized such that the Arduino-based controller can be swapped out with a S# model of the control software or implementations for other microcontroller architectures.

integrate the Arduino software into a S# model: The first possibility is to compile the C code for x64 hardware as a dynamic library that is then loaded into the S# analysis process; instead of the Arduino SDK, a custom library must be linked with the Arduino software that implements functions like `digitalRead` and manages the interactions with the S# framework. On the S# side, a new controller implementation must be developed that marshals the sensor and actuator data between the S# model and the C code using .NET's platform invocation services [110]. Alternatively, the C program can be run on actual Arduino hardware. In that case, a S# component must be developed that communicates with the Arduino board. While both approaches require some work when implementing the actual hardware and software components, the additional effort further increases the confidence in the soundness, adequacy, and safety of the actual system. For existing systems, an integration into a S# model might be hard to achieve retroactively, however, the benefit is potentially large and the required effort might be acceptable when planned for from the beginning. Chapter 8 discusses an analysis and testing approach for self-organization mechanisms and adaptive robot systems that takes advantage of S#'s software- and hardware-in-the-loop analysis capabilities.

Visualizations. As S# models are regular C# programs, their states can be visualized using various UI frameworks available for C#. For the case studies, for example, the Windows Presentation Foundation [147] is used as it is a flexible framework with decent designers and Visual Studio integration. Figure 6.9 shows two screenshots of the visualizations of the pressure tank and height control case studies. Both visualizations are interactive, allowing mouse or touch input to activate faults on command, to spawn high and overheight vehicles, or to change the vehicles' lanes. Additionally, visual replays of counterexamples are also supported. Visualizations only indirectly use the `Simulator` class provided by the S# framework. Instead, they build upon `RealTimeSimulator` instances that wrap `Simulator` instances as shown in lines 1 and 2 of the example below,



(a) A replay of a DCCA counterexample showing that activations of ¬is full and ¬timeout lead to a tank rupture, indicated by the lightning bolt through the tank. The lightning bolts at the pressure sensor and the timer indicate the fault activations. The controller still thinks that it should continue pumping.



(b) An interactive simulation of the height control case study. To the right, a high vehicle is about to enter the tunnel on the left lane while an overheight vehicle approaches the main control's sensors. Vehicles can be spawned interactively using the buttons on the left.

Figure 6.9. The S# models of the pressure tank (left) and height control (right) case studies are visualized using the Windows Presentation Foundation; the visualizations are available in the S# repository [101]. The visualizations can be started, stopped, forwarded, rewound, slowed down, or accelerated while either replaying a model checking counterexample or running an interactive, exploratory simulation during which faults are interactively activated on command.

hence the basic simulation and counterexample replay functionality provided by the `Simulator` class is reused. The `RealTimeSimulator` represents an execution engine for interactive visualizations that delays invocations of the `SimulateStep` method in accordance with the model's step time Δt. For example, $\Delta t = 1s$ for the pressure tank case study, so the `RealTimeSimulator`'s `Run` method invoked in line 6 internally calls `SimulateStep` on the underlying `Simulator` instance once per second as specified in line 2. Whenever the simulation updated the state of the simulated `model` retrieved on line 3, the `ModelStateChanged` event is raised, allowing the visualization to be updated based on the new state of `model` as indicated by the event handler lambda function attached to the event in line 5.

```
1 var simulator = new Simulator(new Model());
2 var realTimeSimulator = new RealTimeSimulator(simulator, stepTime: 1000 /* ms */);
3 var model = (Model)realTimeSimulator.Model;
4
5 realTimeSimulator.ModelStateChanged += () => /* update the visualization */;
6 realTimeSimulator.Run();
```

Replaying Counterexamples. To replay a model checking counterexample, an instance of the `Counterexample` class must be obtained from the results returned by an `SSharpChecker` instance as illustrated in Section 6.3.1. Alternatively, a previously saved counterexample can be loaded from disk as shown in line 1 of the following example. Subsequently, a `Simulator` instance is created for the `Counterexample` instance in line 2 followed by the retrieval of the simulated instance of the pressure tank case study

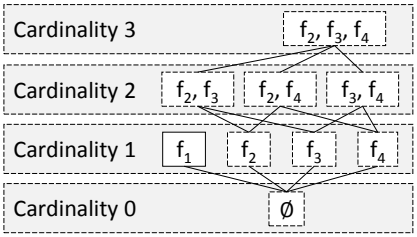
model in line 3. The `SimulateStep` method is used in the loop declared on line 5 to iterate through all steps the counterexample consists of. Each iteration of the loop body consecutively makes the individual model states available that led to the violation of the analyzed formula the counterexample was created for. Thus, relevant state information could be printed to the console, a visualization could be updated, or a debugger could be attached to step through the individual micro steps of the counterexample.

```
1 var counterexample = Counterexample.Load(/* file name */);
2 var simulator = new Simulator(counterexample);
3 var model = (Model)simulator.Model;
4
5 while (simulator.SimulateStep())
6 {
7    // ...
8 }
```

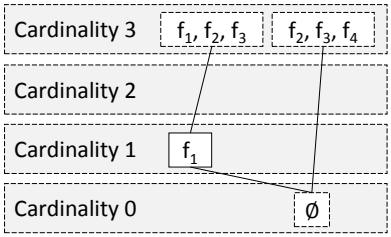
6.4 Safe Fault Sets Heuristics

By default, S# uses a bottom-up DCCA implementation that only takes advantage of the monotonicity of the criticality property. If the search was conducted top-down, the monotonicity of the safe fault set property could be taken into consideration instead. As most safety-critical systems have rather small minimal critical fault sets, the bottom-up search strategy is usually preferable as it has to carry out fewer checks. But as soon as the cardinalities of the minimal critical fault sets increase or the number of minimal critical fault sets for a hazard becomes larger, a combination of both search strategies might be beneficial. The naïve approach of randomly checking very large fault sets most likely does not decrease analysis times, however: Many of the larger fault set are likely to be critical and checking them might be expensive as the fault removal optimization becomes less effective the larger the fault sets become; for larger fault sets $\Gamma \subseteq F(K)$, only a few irrelevant faults $F(K) \setminus \Gamma$ can be removed from the model.

Instead of randomly checking large fault sets for criticality, the S# framework supports safe fault sets heuristics when conducting DCCAs, thereby combining the strengths of both the bottom-up and the top-down search strategies. While DCCAs are still primarily conducted bottom-up, the heuristics are allowed to suggest fault sets of higher cardinalities that are likely to be safe; if the suggested sets are indeed safe, a potentially large number of criticality checks can be avoided compared to the pure bottom-up search strategy: If a suggested fault set $\Gamma \subseteq F(K)$ of cardinality n in fact turns out to be safe, up to $2^n - 1$ checks can be avoided as all $2^n - 1$ subsets of Γ are immediately known to be safe as well. In the general case, however, the heuristics cannot be absolutely sure that a suggested fault set is indeed safe, therefore all suggestions must be checked for criticality nevertheless. Consequently, heuristics always try to guess fault sets that are as large as possible while still being reasonably sure that the suggestions are indeed safe. If a critical set is suggested, additional work is introduced without speeding up DCCA, potentially even slowing the process down as a critical fault set is checked that is likely to be non-minimal and would thus have been skipped by the bottom-up search. Figure 6.10 illustrates the effects of heuristics on the number of fault sets that have to be checked for criticality.



(a) The bottom-up search strategy quickly finds the minimal critical fault set $\{f_1\}$, allowing it to skip the checks for all fault sets of cardinality $n \geq 2$ that contain f_1 due to the monotonicity of the criticality property. However, a large number of safe fault sets has to be checked as the bottom-up search cannot take advantage of the monotonicity property of safe fault sets. Overall, a total of nine criticality checks have to be performed to complete the DCCA.



(b) A heuristic might suggest the fault sets $\{f_1, f_2, f_3\}$ and $\{f_2, f_3, f_4\}$, assuming them to be safe for the analyzed hazard. As the former is critical, the suggestion is useless except that $\{f_1, f_2, f_3, f_4\}$ is immediately known to be critical as well. In the worst case, such useless suggestions can noticeably slow down analyses. The latter suggestion, however, is indeed a safe fault set, immediately resulting in all of its subsets to be classified as safe. A bottom-up search is started for the remaining fault sets, i.e., $\emptyset$ and $\{f_1\}$ in this case. Consequently, only four criticality checks have to be performed to complete the DCCA thanks to the heuristic.

Figure 6.10. Comparison of the number of fault sets that have to be checked for criticality using only the bottom-up search strategy (left) versus the bottom-up search strategy amended with top-down searches by heuristics (right). The model is assumed to contain four faults f_1, f_2, f_3, and f_4, but only $\{f_1\}$ is assumed to be a minimal critical fault set for the analyzed hazard.

Heuristics generally try to guess maximal safe fault sets: These fault sets are the dual of minimal critical ones in the sense that they cannot be enlarged with any additional faults without making them critical. Formally:

Definition 6.3 (Maximal Safe Fault Sets)**.** *A safe fault set* $\Gamma \subseteq F(K)$ *is* maximal *if no proper superset* $\Gamma' \supsetneq \Gamma$ *is safe.*

Consequently, maximal safe fault sets represent a frontier of large fault sets behind which all other fault sets are known to be critical. Ideally, all suggestions made by a heuristic would be maximal safe fault sets for two reasons: Firstly, a suggestion that turns out to be critical is likely useless if not outright counterproductive, as a large fault set has to be checked for criticality without providing any meaningful information; the only exceptions are suggested fault sets that are minimal critical as these would eventually have to be checked anyway later on. Secondly, suggested sets of higher cardinalities have a larger potential for reductions in the number of safe fault sets that no longer have to be checked. However, maximal safe fault sets can typically only be guessed; if they were known, the minimal critical fault sets could be computed from them and no DCCA would have to be conducted in the first place. Therefore, heuristics are imperfect, sometimes suggesting critical fault sets or safe ones that are not maximal.

In Figure 6.10, for example, there only is one maximal safe fault set: $\{f_2, f_3, f_4\}$. It is the only set of the 16 possible fault sets that is safe and that only has critical supersets. When there are more minimal critical fault sets and, in particular, minimal critical fault sets of larger cardinalities, there typically are multiple maximal safe fault sets that are partially overlapping. A case study-independent heuristic would thus be the computation of all

	Initial DCCAs w/o Heuristic 1		Initial DCCAs w/ Heuristic 1		Subsequent DCCAs w/ Heuristic 1	
	Time	# Checks	Time	# Checks	Time	# Checks
Height Control						
Collisions	14.2s	325	3.4s (4.2x)	53 (6.1x)	1.0s (14.2x)	5 (65.0x)
False Alarms	34.8s	261	2.9s (12.0x)	42 (6.2x)	1.9s (18.3x)	6 (43.5x)
Railroad Crossing						
Potential Collisions	1.2s	11	1.2s (1.0x)	11 (1.0x)	0.4s (3.0x)	8 (1.4x)
Hemodialysis Machine						
Contamination	26.7s	243	6.5s (4.1x)	54 (4.5x)	6.4s (4.2x)	11 (22.1x)
Dialysis Failure	0.9s	14	0.9s (1.0x)	14 (1.0x)	0.8s (1.1x)	7 (2.0x)

Table 6.6. Evaluation of the effectiveness of heuristic 1 for the height control, railroad crossing, and hemodialysis machine case studies. The self-organization case studies require more specialized heuristics as discussed in Section 8.2 as heuristic 1 cannot effectively handle the high number and large cardinalities of their minimal critical fault sets. Initial DCCAs are conducted either with or without the heuristic; in the former case, the heuristic makes its suggestions based on a partially computed set of minimal critical fault sets. Subsequent DCCAs can reuse previous analysis results; if the analyzed models remain unchanged between the initial and subsequent DCCAs, the fault sets suggested by the heuristic are guaranteed to be safe. Overall, heuristic 1 is highly effective for subsequent DCCAs but also speeds up initial ones noticeably.

maximal safe fault sets based on already known or at least assumed minimal critical ones; two case study-dependent heuristics are introduced in Section 8.2. More formally, the maximal safe fault sets heuristic takes advantage of DCCA completeness as follows:

Heuristic 1 (Maximal Safe Fault Sets Heuristic)**.** *For a known or assumed set $\Lambda_H \subseteq 2^{F(K)}$ of minimal critical fault sets for hazard H and fault-aware Kripke structure K, the fault sets $\{F(K) \setminus \{f_1, \ldots, f_n\} \mid f_i \in \Gamma_i$ for all $1 \leq i \leq n$ where $\Lambda_H = \bigcup_{i=1}^{n} \Gamma_i\}$ are likely to be maximal safe fault sets for H.*

If Λ_H is already known from previous analyses, the maximal safe fault sets heuristic can significantly speed up DCCAs. Assuming that no changes have been made to the analyzed model that completely or partially invalidate a previously obtained set Λ_H, all suggestions made by the heuristic indeed turn out to be safe fault sets. On the other hand, if changes to the model have an effect on Λ_H, these often only add, remove, or change some of the minimal critical fault sets, thus there still is a good chance that the heuristic succeeds in suggesting at least some safe fault sets. Moreover, the maximal safe fault sets heuristic can also be used to speed up initial DCCAs for which no minimal critical fault sets are already known. In this case, the heuristic can be instructed to suggest the maximal safe fault sets as soon as all minimal critical fault sets up to a specific cardinality are known. That is, the heuristic can also make its suggestions based on only a partially known set Λ_H, in which case, however, the suggestions can be wrong. Table 6.6 evaluates the potential gains achieved with the heuristic.

The S# framework enhances the DCCA algorithm illustrated by Algorithm 2 on page 112 by incorporating the heuristics for safe fault sets. As heuristics are often case study-specific such as the ones for self-organizing resource-flow systems introduced in Chapter 8, the S# framework provides the following interface for custom heuristics:

```
interface ISafeFaultSetsHeuristic
{
   void Augment(int cardinality, LinkedList<FaultSet> setsToCheck);
   void Update(LinkedList<FaultSet> setsToCheck, FaultSet checkedSet, bool isSafe);
}
```

The set of heuristics to be used must be specified before a DCCA is started, allowing each case study to benefit from those heuristics that are most effective for it. That is, only those heuristics should be specified for a DCCA that suggest many safe fault sets of high cardinalities. S#'s DCCA implementation calls the `Augment` method for each level of fault set cardinality, passing along a linked list of fault sets that have to be checked next. The heuristics are free to add any fault sets to the list that they deem safe, regardless of the cardinality. Additionally, the list is always checked from front to back, that is, the order of inserts affects the effectiveness of the suggestion. For example, a large fault set Γ should generally be added to the front of the list, because when it is safe, all of its subsets $\Gamma' \subsetneq \Gamma$ following Γ in the list can be skipped as they are then already known to be safe as well. For some heuristics, however, it is occasionally useful to add suggestions at different locations. Heuristics can also remove sets from the list, in which case S# ensures that the analyses remain complete, that is, heuristics in general cannot remove sets they did not previously add themselves. The use of the `LinkedList<T>` data type is necessary for reasons of efficiency, as insertions and removals are always $O(1)$ compared to `List<T>` where they are $O(n)$ in the worst case. When millions of fault sets have to be checked for criticality, the use of `List<T>` introduces unacceptable element copying overhead, taking up to 90% of the entire analysis time in some cases. The `Update` method is used to inform the heuristics about safe and critical fault sets, allowing them to adjust their future suggestions based on the quality of their past ones.

For example, to set up a S# DCCA for the hazard of false alarms in the height control case study that reuses previous analysis results, the following code can be used. In line 4, an instance of the `MaximalSafeFaultSetsHeuristic` is added to the `SafetyAnalysis` instance created in line 3, which is subsequently used when the minimal critical fault sets are computed in line 6.

```
1 var model = /* ... */;
2 var result = /* result from previous DCCA */;
3 var analysis = new SafetyAnalysis();
4 analysis.Heuristics.Add(new MaximalSafeFaultSetsHeuristic(model,
      result.MinimalCriticalSets));
5
6 result = analysis.ComputeMinimalCriticalSets(model, model.FalseAlarm);
```

6.5 Related Work

Execution versus Transformation. Many safety analysis tools such as VECS, the Compass toolset, or AltaRica [15, 135, 151] rely on the standard approach of model transformations to use model checkers like SPIN or NuSMV [38, 98]. By contrast, S# unifies simulations, visualizations, and fully exhaustive model checking by executing the C# models with consistent semantics. Consequently, no model transformations changing the level of expressiveness are necessary, avoiding significant implementation complexity while retaining competitive model checking efficiency. S# only has to execute

C# code instead of understanding and transforming it, supporting most C# language features without any additional work; transformations, by contrast, would require large parts of the .NET virtual machine to be encoded for model checking or to forgo many higher level C# features such as virtual dispatch or lambda functions. The two main challenges of model checking S# models are efficient state serialization and efficient handling of nondeterminism. The algorithm that allows ChoiceResolver to handle and track all combinations of nondeterministic choices, however, only requires around 40 lines of C# code. Generating low overhead serialization methods, by contrast, is more involved, taking about 700 lines of C# code to generate the methods at run time. The model transformations carried out by the Compass toolset [151], by contrast, require several thousand lines of Python code; due to the very different levels of expressiveness, the resulting NuSMV models can become unreadable. Additionally, the toolset provides its own AADL parser with a grammar description file of 700 lines. The necessary code for parsing and semantic analysis also introduce significant complexity with several thousand lines of Python code. S#, by contrast, simply reuses C#'s standard compiler and adds roughly 600 lines for additional semantic analyses and 1600 lines of C# code for the necessary code transformations. These additions deal with ports and fault effects, i.e., those things that lie at the core of the S# modeling language. The S# compiler therefore does concern itself with standard compiler construction problems such as expression parsing, good error messages and recovery, or type checks that are of low theoretical interest for model-based safety analysis. Moreover, model debugging and testing are inherently supported without any additional implementation effort that would be required by a transformation-based analysis approach.

State Storage Efficiency. For the pressure tank, height control, railroad crossing, and hemodialysis machine case studies, serialization causes only around 5% of overhead during the entire model checking process; the self-organization case studies have higher overhead as they have to serialize object references, which is more time-consuming due to the necessary dictionary look-ups. The serialized states are smaller than the state vectors of a hand-optimized SPIN models of the height control and railroad crossing case studies, taking only 12 instead of 24 bytes per state. Consequently, the S# framework in general halves the memory requirements for model checking compared to SPIN. Symbolic model checking techniques, by contrast, do not consider states individually, only storing relevant sets of states; they thus usually have lower memory requirements than explicit-state model checkers.

Explicit-State Model Checking with SPIN. In the worst case of valid formulas, S# and LTSmin have to enumerate the model's entire state space, taking 53.2 seconds for the height control case study using a single CPU core only. SPIN, by contrast, takes 553 seconds to check a hand-optimized, non-modular SPIN version of the model that semantically corresponds to the S# version. On a quad-core CPU, LTSmin achieves a speedup of 3.7x, bringing the analysis time down to 14.2 seconds whereas SPIN scales by a factor of 1.5x only. For a version of the railroad crossing case study with increased resolution, the S# model takes 126 seconds to enumerate all states, whereas a manually created SPIN model takes 114 seconds, i.e., it is slightly faster. With multi-core model checking, however, the S# framework achieves a speedup of 3.8x, bringing analysis

time down to 33.2 seconds compared to SPIN's 77.9 seconds due to its suboptimal multi-core scaling. The main reasons for S#'s superior performance for the height control case study is that fault-awareness and the fault activation optimizations described in Section 6.2.2 can only be partially encoded into the SPIN models; they can only be fully supported by changing SPIN's core model checking algorithm. S# is therefore able to ignore irrelevant fault activations more efficiently than SPIN, causing it to compute less transitions while still finding all reachable states.

Symbolic Model Checking with NuSMV. For the height control case study, symbolic analysis with NuSMV is faster than using S#: For a hand-written, very low-level and non-modular NuSMV model that is approximately equivalent to the S# model, the entire state space is generated almost instantly compared to the 14.2 seconds it takes the S# framework to do the same. On the other hand, the railroad crossing case study is checked 2.7 times faster by S# than by NuSMV. The relative efficiency of explicit-state and symbolic model checking is therefore case study-specific and independent from S#; in general, highly nondeterministic models seem to profit more from symbolic techniques. Generally, analysis times of explicit-state model checkers are more predictable than those of symbolic model checkers, as the latter rely on heuristics that allow them to consider combined sets of states [13, 38]. For the height control case study, these heuristics are highly effective, but changes to the model that reorder variables, for instance, could have unpredictable negative influences on the effectiveness of these heuristics, suddenly slowing down NuSMV significantly. Moreover, using input variables to encode fault-awareness does not improve NuSMV's analysis efficiency for the height control and railroad crossing case studies. Consequently, NuSMV does not seem to profit from fault-aware modeling and specification. In general, NuSMV achieves its efficiency through the low level of expressiveness of its models, in particular restricting them to synchronous variable updates only.

Language Expressiveness versus Analysis Efficiency. S# models have a much higher level of expressiveness than either SPIN or NuSMV models, allowing variant modeling and analysis in a way that is not supported by either model checker directly. In particular, the models of the self-organization case studies, cf. Chapter 8, reach a level of complexity that necessitates the use of high-level modeling languages such as S# with good model development, debugging, and testing support. Additionally, S#'s explicit support for fault modeling guarantees conservative extension, while SPIN or NuSMV, for instance, cannot give this guarantee at all. While S# is more efficient than SPIN due to its fault optimizations and multi-core model checking capabilities, the increase in analysis time compared to NuSMV for some case studies seems acceptable given the step-up in modeling flexibility, expressiveness, and fault modeling adequacy. A transformation of S# models to NuSMV models is not efficiently feasible, as the models would have to be symbolically executed during the transformation which either has exponential complexity or requires additional variables [64] that NuSMV cannot efficiently cope with. The underlying reason for the complexity of such transformations is S#'s support for sequential variable writes within a macro step, whereas NuSMV only supports non-conflicting, synchronous updates to variables within a step. Consequently, this mismatch between the models of computations gives S# advantages in modeling

expressiveness at the cost of symbolic analysis efficiency, highlighting the need for fault-aware modeling and specification in order to bring explicit-state model checking of executable models to a competitive level with symbolic analysis techniques.

Other Safety Analysis Tools. VECS [135] is a low-level abstraction of the NuSMV and Prism input languages, i.e., its level of expressiveness is very similar, thus the remarks on symbolic model checking with NuSMV also apply to VECS. The Compass toolset [151], on the other hand, has a higher-level modeling language, albeit still with a model of computation similar to NuSMV. In general, it is therefore hard to achieve fair comparisons between these tools and symbolic analysis techniques on the one hand and S# and explicit-state model checking on the other hand. For instance, it takes about 740 lines to create a scaled-down Compass model of the railroad crossing case study that is semantically similar to the S# version written in approximately 400 lines of C# code. Compass performs a safety analysis based on the FSAP/NuSMV platform [24, 26] that is equivalent to DCCA in 21 minutes using NuSMV instead of the 1.2 seconds it takes S# to do the same. Of course, the comparison is unfair as forcing Compass semantics onto S# might likewise slow down analyses.

Software Model Checking. The S# framework is not a software model checker such as MoonWalker, Java Pathfinder, or Zing [1, 4, 194] even though the underlying techniques are similar: Programs are executed within the context of an on-the-fly model checking algorithm [124]. The primary application of software model checking is the identification of data races and deadlocks in concurrent programs. In order to be able to do so, the assumption of zero execution time underlying all S# models is invalid for software model checking, as the precise timing of the individual operations, i.e., the micro steps, carried out by the different parts of the analyzed program are indeed relevant. Furthermore, software model checking is mainly concerned with functional correctness instead of safety, that is, the analyzed programs do not contain any plant components or faults that are required for safety analyses.

Summary and Outlook. The model execution approach presented in this chapter unifies various kinds of analyses that are useful for different purposes during the development of safety-critical systems. For example, model checking allows for formal safety analyses based on DCCA whereas simulations and visualizations enable debugging or visual replays of model checking counterexamples. Since all of these analyses are based on model execution, their consistency is guaranteed and no complicated model transformations are required. Compared to other safety analysis tools, the implementation of the S# framework is therefore less complex, but thanks to its anchorage in the C# and .NET world, it can take advantage of all of the tooling available to regular .NET developers. The formal techniques introduced in Chapter 5 are inherited by executable models and the S# framework in particular, as model execution can be used to construct fault-aware Kripke structures. The S# framework, however, never explicitly creates fault-aware Kripke structures during analyses of a model, instead using on-the-fly model checking and fault activation minimization techniques to further speed up analyses. Chapters 7 and 8 show the applicability of the formal techniques and their integration into the S# framework using S# models of the height control and the self-organization case studies, respectively. The latter chapter also introduces additional DCCA heuristics and discusses a modeling and analysis approach for run time safety analyses of self-organizing systems.

Summary. The height control case study is modeled with S# and analyzed with DCCA. As the original design of the case study is functionally incorrect, multiple design alternatives are discussed, modeled, and analyzed. S#'s flexible model composition capabilities are leveraged to conveniently instantiate all valid combinations of system design variants, assembling together the orthogonally modeled variants of different controller components in various ways. DCCA's analysis efficiency is evaluated using both hazards of the case study as well as all combinations of modeled design alternatives.

Publications. The S# model of the height control case study is published in [84]. It can be obtained from the S# repository [101], including a S#-based visualization.

7

Design Time Analysis of the Height Control Case Study

The height control case study introduced in Chapter 2 is a classical safety-critical system in the sense that its structure and behavior is fully determined during development. Safety analyses can therefore be completely carried out during system design as opposed to self-organizing systems for which run time analyses are necessary as discussed in Chapter 8. The first safety analysis of the height control case study was conducted in 2002 by Ortmeier et al. [158] using a mixture of Fault Tree Analysis and model checking of unstructured, non-modular, and low-level transition systems. Subsequent analyses were based on DCCA instead [159], still using a rather low-level modeling formalism. These analyses concluded that the original specification of the case study is inadequate, as both false alarms and collisions can in fact happen without a single fault being activated beforehand, that is, the empty set of faults is minimal critical for both hazards. Several design alternatives that add additional sensors, for instance, were proposed to fix the problem [158], necessitating additional safety analyses to check for newly introduced safety issues. However, each analyzed variant required manual changes to a copy of the model, making it hard to ensure consistency between them while also introducing significant modeling overhead. S#'s support for variant modeling and automated composition of different design alternatives, on the other hand, can be

leveraged to more conveniently model the different and partially orthogonal variants in a modular way, automatically composing all combinations together for fully automated safety analyses based on DCCA.

In the early stages of development, it is often unknown how a system is best designed in a way that enables it to fulfill its intended functionality without any unacceptable safety risks. Several different design variants are often conceivable that have to be evaluated for their development and production costs, correctness, adequacy, and safety, among other things [131, 161, 186]. Formal safety analyses can be carried out on models of the system under development before the system is actually built, allowing its safety to be assessed while it is still comparatively easy and cheap to make modifications to the system's design. Such analyses typically unveil unacceptable safety risks that must subsequently be alleviated by introducing additional safety measures into the system design, requiring a new round of safety analyses to check whether the changes had their intended effects. Consequently, the minimal critical fault sets must show that the changes indeed improve safety to a degree that justifies the development and production costs of the additional safety precautions.

Similar to design exploration, safety-critical product lines [16, 191] also consist of multiple system variants developed at design time, albeit designed and distributed based on the specific needs of a customer. For example, cars are often built in this way, allowing the customers to purchase additional safety-critical features at an extra cost such as autonomous cruise control, for instance [115, 190]. Separate safety analyses have to be carried out for all of the different product configurations to ensure their safety. But as there often are many different orthogonal features that can be configured into a system, the amount of safety analyses that have to be conducted increases exponentially. While S# provides modeling language features based on C#'s object-oriented concepts to support variability modeling, that is, the specification of different designs and product features, it does not yet support integrated, simultaneous analyses of multiple different system variants [191]. However, to some extent it is possible to reuse analysis results from one variant when analyzing the next one using the maximal safe fault sets heuristics to increase analysis efficiency [46]. Consequently, S# provides some first steps towards modeling and analyzing safety-critical product lines, but full integration of safety analysis and product line engineering with formal methods requires the development of some additional analysis techniques.

The following presents a S# model of the height control case study adhering to the systematic modeling approach presented in Chapter 3. The model of the original case study design introduced in Section 7.1 is extended to incorporate several design alternatives for the case study's controller components in Section 7.2. Section 7.3 composes the different system designs together by taking advantage of S#'s flexible model composition capabilities based on C#'s expressiveness and the standard reflection facilities provided by the .NET run time and its base library. Subsequently, Section 7.4 presents the results of the DCCAs automatically carried out by the S# framework for all modeled design variants, discussing the flaw in the case study's original design. The entire S# model of the case study is available in the S# repository [101], including an interactive, S#-based visualization.

7.1 Modeling the Original Design

In the height control case study, the vehicles constitute the plants that are observed and controlled by the three subcontrollers the overall controller consists of. The two hazards that are analyzed can be specified over the vehicles' positions and lanes as well as the fact whether the tunnel is closed, that is, whether the traffic lights closing the tunnel are switched to red. Consequently, a collision occurs when an overheight vehicle reaches the tunnel entrance on the left lane, whereas a false alarm occurs when the tunnel is closed even though there is no overheight vehicle on the left lane in the entire height control area. Neither hazard could accurately be formulated without the vehicles being part of the model, thus showing the necessity to model the vehicles, or more generally, the controlled plants, for formal safety analyses. For false alarms, the necessity of the vehicle models is particularly obvious: False alarms are control failures that the height control is unaware of, otherwise it would not have mistakenly closed the tunnel. A false alarm can thus never be accurately detected by the control system alone even if there are additional safety measures that try to prevent them; after all, such detection capabilities also make use of hardware components that could fail as well.

The structure of the height control model is shown by the block definition diagram in Figure 7.1. Even though only the original system design is considered in this section, the model is already prepared for the additional design variants in Section 7.2: There are three abstract blocks representing the common parts of all variants of the pre, main, and end controls. These blocks mostly define the structural aspects while they themselves specify almost no behavior. Consequently, it is possible to replace different variants with each other as long as all of these variants adequately implement the ports declared by the abstract blocks and other parts of the model do not depend on any concrete subcontroller types. The original designs are structurally very similar to the abstract blocks as seen in Figure 7.2, with some of the other variants adding additional sensors, for instance, as discussed in Section 7.2. The original design variants of the subcontrollers model the height control behavior as originally envisioned; in particular, the MainControlOriginal uses a counter to track the number of overheight vehicles that are assumed to be in its observed area. The internal block diagram in Figure 7.3 shows a model instance with PreControlOriginal, MainControlOriginal, and EndControlOriginal block instances, therefore representing the original case study design. The VehicleDetector and TrafficLight instances are modeled as nested ports as indicated by the «port» stereotype in the blocks' definitions shown in Figure 7.1.

Another abstraction is introduced through the abstract VehicleDetector block that both LightBarriers and OverheadDetectors derive from. There are two main reasons for this modeling decision: On the one hand, modularity increases as additional sensor types other than LightBarrier and OverheadDetector could be used transparently as long as they adhere to VehicleDetector's semantic contract; however, none of the analyzed design variants currently takes advantage of this possibility. On the other hand, misdetections and false detections can be modeled exclusively in terms of VehicleDetector, demonstrating S#'s fault modeling capabilities by describing the faults' effects once for all types of sensors considered in the case study. The individual parts of the case study model as well as the faults are explained in detail in the following.

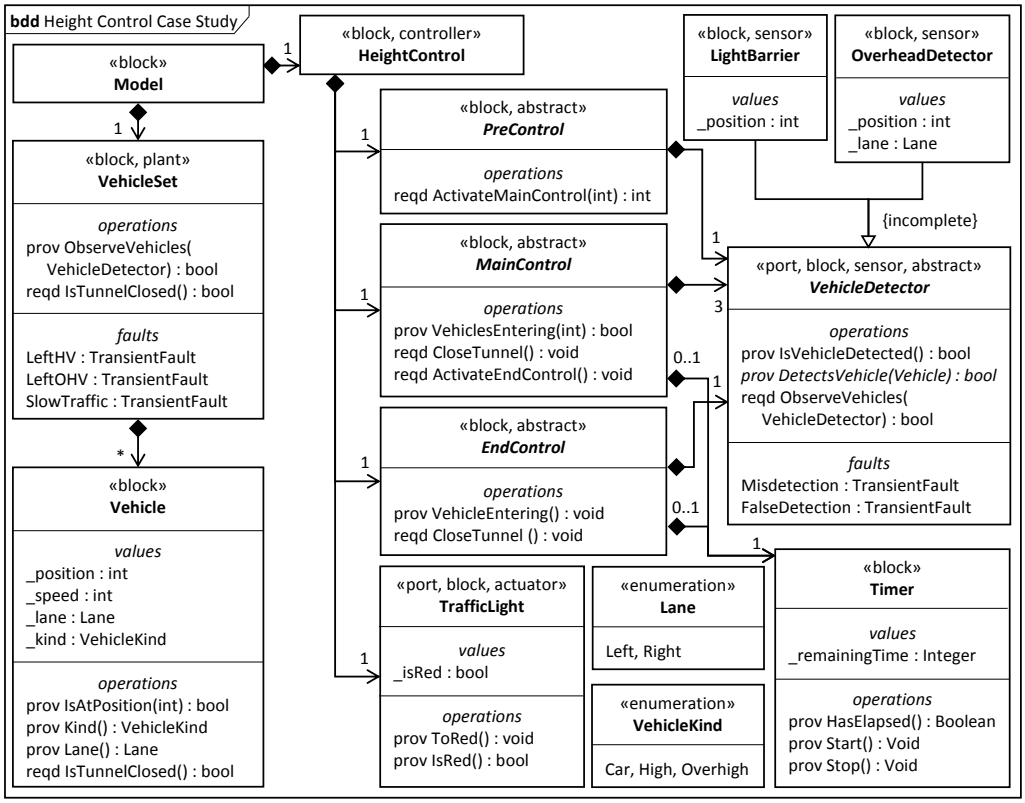


Figure 7.1. An incomplete block definition diagram of the height control case study omitting all fault effects for presentational clarity. The model consists of the plant, i.e., the VehicleSet, and the HeightControl controller block. The two sensor types are abstracted away behind a common VehicleDetector block. The TrafficLight is the only actuator in the case study, signaling tunnel closures. The HeightControl is subdivided into three abstract subcontrollers to support variant modeling as discussed in Section 7.2. All controller designs share some common VehicleDetector subcomponents that they use to observe the vehicles. Additionally, MainControl and EndControl use the Timer block to automatically deactivate themselves after some time.

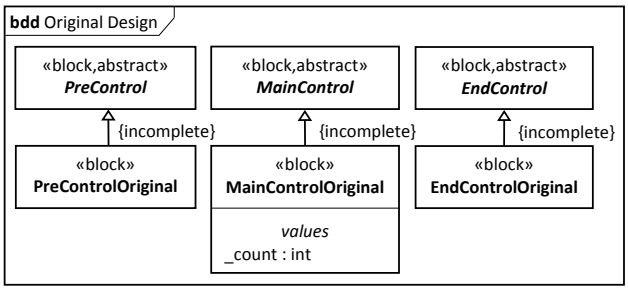


Figure 7.2. The original designs of the subcontrollers are closely related to the abstract base blocks from Figure 7.1, only specifying their respective behavior without introducing any new associations or operations. The original design of the MainControl adds a new state variable _count to track the number of overheight vehicles that are assumed to be present in its observed area, that is, the vehicles that are in between the PreControl's and MainControl's vehicle detectors.

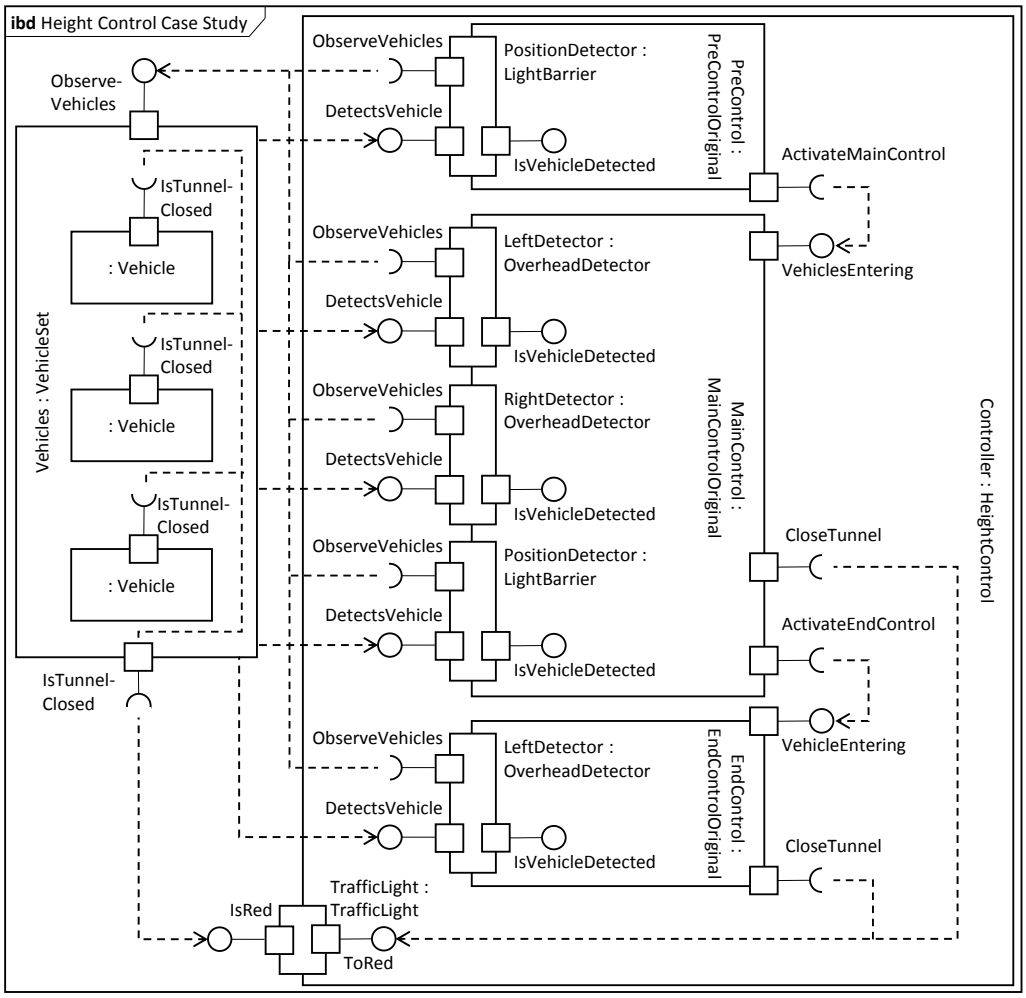


Figure 7.3. A partial internal block diagram showing the connections between the Vehicles and the Controller of the original case study design. The Timer instances contained in MainControl and EndControl are omitted for reasons of brevity as their effects in the computations of the subcontrollers' provided ports cannot be shown in an internal block diagram anyway. The connections between the subcontrollers show how information about overheight vehicles propagates through the system: When the PreControl detects an overheight vehicle, MainControl is informed via its VehicleEntering provided port; whenever such a vehicle leaves the main control area on the right lane, EndControl is activated through its VehicleEntering provided port. The CloseTunnel required ports of the `MainControl` and the `EndControl` indicate whether a tunnel closure is necessary, invoking the ToRed port of the TrafficLight. The Vehicle instances contained in the VehicleSet stop their approach towards the tunnel whenever the TrafficLight's IsRed port returns true. The VehicleSet is responsible for broadcasting the TrafficLight's state to all of its Vehicles, as the model supports an arbitrary number of Vehicle instances; in the following, three instances are used, two of which are of kind VehicleKind.Overhigh and the remaining one is of kind VehicleKind.High (not shown in the figure). Regular cars are not considered during formal analyses as they are unobservable by the VehicleDetectors; they are useful for visualization purposes, however. While the ObserveVehicles ports are connected, the VehicleSet directly calls the DetectsVehicle ports on the VehicleDetector instances passed to ObserveVehicles as explained later.

7.1.1 Abstract Vehicle Modeling

The S# model of a vehicle shown below allows it to drive on either of the two lanes with a certain speed, chosen nondeterministically, stopping whenever the tunnel is closed. Initially, a vehicle drives on the right lane and its speed as well as its position are implicitly set to zero; the kind is specified via the constructor. The position is range-restricted to fall within $[0; \texttt{Model.TunnelPosition}]$ with the overflow behavior "teleporting" the vehicle back to its initial position whenever it reaches the tunnel. The speed is `[Hidden]` as in all macro steps, the vehicle always chooses a new speed in the Update method before the speed is used in other micro steps. The `IsAtPosition` provided port hides the effects of position and speed discretization from other components: In order to determine whether a vehicle is at position p, the position at the beginning of a macro step is computed by subtracting the current speed from the current position, subsequently ensuring that the resulting position is smaller than p and the current position is greater than or equal to p; if that is the case, the vehicle has passed p in the current macro step. The `Update` method nondeterministically decides when the vehicle starts approaching the tunnel, that is, as long as the vehicle's position is zero, it is free to remain where it is. Once it decides to approach the tunnel, the `Update` method nondeterministically chooses a new speed and subsequently updates the position if the tunnel is not closed. Due to the road layout, the nondeterministic lane changes are only possible before the end control area is reached.

```
class Vehicle : Component
{
   [Hidden] private int _speed;

   public Lane Lane { get; private set; } = Lane.Right;
   public VehicleKind Kind { get; }

   [Range(0, Model.TunnelPosition, OverflowBehavior.WrapClamp)]
   public int Position { get; private set; }

   public Vehicle(VehicleKind kind)
   {
      Kind = kind;
   }

   public bool IsAtPosition(int pos) => Position - _speed < pos && Position >= pos;
   public extern bool IsTunnelClosed { get; }

   public override void Update()
   {
      if (Position == 0 && Choose(true, false))
         return;

      if (IsTunnelClosed)
         return;

      _speed = Choose(Model.MinSpeed, Model.MaxSpeed);
      Position += _speed;

      if (Position < Model.EndControlPosition)
         Lane = Choose(Lane.Left, Lane.Right);
   }
}
```

The following `VehicleSet` component is introduced in order to allow the overall case study model to be transparently instantiated with an arbitrary amount of vehicles. During model execution, however, the number of vehicles remains fixed. The `Vehicle` references stored in the `Vehicles` array never change, hence field and element hiding is used to optimize the state vector. Each vehicle's `IsTunnelClosed` required port is forwarded to the set's `IsTunnelClosed` required port using an intermediate `ForwardIsTunnelClosed` provided port as S# does not natively support such port forwardings. In the `Update` method, all vehicles contained in the set are updated.

Due to the abstractly modeled vehicle behavior, it is necessary to specify state constraints to rule out physically impossible scenarios that negatively affect safety analyses in a completely unrealistic way. In particular, two different `Vehicle` instances could be at the exact same location, allowing them to pass a detector simultaneously such that only one of them is detected. Such a situation cannot happen in real life unless perhaps there is an accident in which case, however, the involved vehicles are unlikely to continue their approach to the tunnel. Consequently, state constraints for all pairs of `Vehicle` instances are introduced below such that they are forced to have pairwise different positions or different lanes. Without these constraints, DCCAs overapproximate the system's safety as the system is assessed to be less safe than it actually is.

```
class VehicleSet : Component
{
   [Hidden(HideElements = true), Subcomponent]
   public Vehicle[] Vehicles { get; }

   public VehicleSet(params Vehicle[] vehicles)
   {
      Vehicles = vehicles;

      foreach (var vehicle in Vehicles)
         Bind(nameof(vehicle.IsTunnelClosed), nameof(ForwardIsTunnelClosed));

      for (var i = 0; i < Vehicles.Length; ++i)
      {
         for (var j = i + 1; j < Vehicles.Length; ++j)
         {
            AddSensorConstraint(Vehicles[i], Vehicles[j], Model.PreControlPos);
            AddSensorConstraint(Vehicles[i], Vehicles[j], Model.MainControlPos);
            AddSensorConstraint(Vehicles[i], Vehicles[j], Model.EndControlPos);
         }
      }
   }

   private bool ForwardIsTunnelClosed => IsTunnelClosed;

   public bool ObserveVehicles(VehicleDetector d) => Vehicles.Any(d.DetectsVehicle);
   public override void Update() => Update(Vehicles);

   public extern bool IsTunnelClosed { get; }

   private void AddSensorConstraint(Vehicle vehicle1, Vehicle vehicle2, int pos)
      => AddStateConstraint(!vehicle1.IsAtPosition(pos) ||
           !vehicle2.IsAtPosition(pos) || vehicle1.Lane != vehicle2.Lane);

}
```

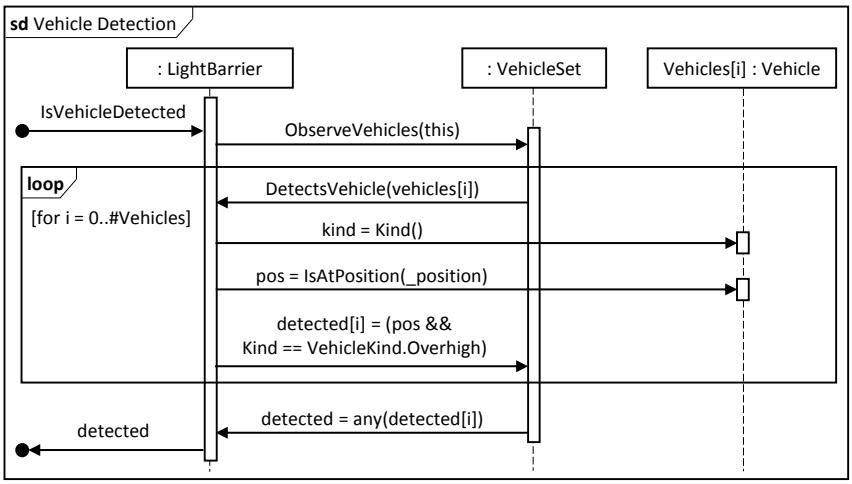


Figure 7.4. A sequence diagram giving an overview of how vehicle detection works within the case study model for LightBarriers; other VehicleDetector-derived blocks work in a similar way: When the LightBarrier's IsVehicleDetected provided port is invoked, it immediately forwards the call to its ObserveVehicles required port, passing its own instance along. As the port is bound to VehicleSet's ObserveVehicles provided port, the set begins to loop through all of its vehicles, calling the LightBarrier's DetectsVehicle port for each Vehicles[i]. The LightBarrier, in turn, gets the VehicleKind from Vehicles[i]'s Kind provided port and checks whether Vehicles[i]'s position matches the LightBarrier's one. If both conditions are satisfied, Vehicles[i] is detected; if any such Vehicles[i] is detected, both ObserveVehicles and IsVehicleDetected return true.

7.1.2 Vehicle Detectors and Traffic Lights

In accordance with the assumption of zero execution time, the resolution of the detectors is assumed to be sufficiently high such that no vehicles are overlooked. To abstract from the number of vehicles, there is a chain of port calls involved when a detector determines whether it detects a vehicle as illustrated by Figure 7.4. A `VehicleDetector` instance thus forwards all invocations of its `IsVehicleDetected` provided port to its `ObserveVehicles` required port as shown below. The `VehicleSet` instance the detector is bound to can then call the detector's abstract `DetectsVehicle` port for each vehicle contained in the set as shown before, with the concrete detector types actually implementing the detection behavior. Detectors therefore neither know nor care about the actual number of vehicles they have to observe. Conversely, the vehicles do not care about the concrete detector types, all in all ensuring maximum modeling flexibility by adhering to the low coupling and high cohesion principles through the use of polymorphism [161].

```
abstract class VehicleDetector : Component
{
   public virtual bool IsVehicleDetected => ObserveVehicles(this);
   public abstract bool DetectsVehicle(Vehicle vehicle);
   public extern bool ObserveVehicles(VehicleDetector detector);
}
```

The `LightBarrier` and `OverheadDetector` components declared in the following are two concrete subtypes of the `VehicleDetector` component. A light barrier spans the entire width of the road, not differentiating between vehicles on the left or on the right

lane. However, it is positioned at a height such that it can only detect overheight vehicles passing by. By contrast, an overhead detector is installed over one of the lanes only, detecting either high or overheight vehicles on that lane. Both detector types therefore declare constructors that allow their positions along the road to be specified; overhead detectors additionally expect the lane that is observed. The constructor parameters are stored in private fields that remain constant during model execution, hence they are declared as `readonly` to exclude them from the state vector. The `DetectsVehicle` provided port is overridden in accordance with the detection capabilities.

```
class LightBarrier : VehicleDetector
{
   private readonly int _position;

   public LightBarrier(int position)
   {
      _position = position;
   }

   public override bool DetectsVehicle(Vehicle v)
      => v.Kind == VehicleKind.Overhigh && v.IsAtPosition(_position);
}

class OverheadDetector : VehicleDetector
{
   private readonly int _position;
   private readonly Lane _lane;

   public OverheadDetector(int position, Lane lane)
   {
      _position = position;
      _lane = lane;
   }

   public override bool DetectsVehicle(Vehicle v)
      => v.Kind != VehicleKind.Car && v.Lane == _lane && v.IsAtPosition(_position);
}
```

The traffic light is the only actuator of the case study. It is modeled as follows, storing whether the light has already been switched to red, also allowing the value to be retrieved via `IsRed`'s getter, i.e., the provided port implicitly declared by the property. The `ToRed` provided port, on the other hand, allows the light to be switch to red, effectively closing the tunnel and preventing all potentially imminent collisions.

```
class TrafficLight : Component
{
   public bool IsRed { get; private set; }
   public void ToRed() => IsRed = true;
}
```

7.1.3 Modular Controller Modeling

The S# model of the abstract `PreControl` block is declared in the following. The component does not declare any behavior, leaving it to the derived types such as `PreControlOriginal`, also shown below, to specify it: The vehicle detector is checked to determine whether an overheight vehicle passes by, invoking the `ActivateMainControl` required port if necessary.

```
abstract class PreControl : Component
{
   [Subcomponent] public readonly VehicleDetector PositionDetector = /* ... */;
   public extern void ActivateMainControl(int vehicleCount);
}

class PreControlOriginal : PreControl
{
   public override void Update()
   {
      Update(PositionDetector);

      if (PositionDetector.IsVehicleDetected)
         ActivateMainControl(vehicleCount: 1);
   }
}
```

The `MainControl` component declared below is the most complex subcontroller of the case study. It uses three `VehicleDetectors` to observe the vehicles in its area and a timer to deactivate itself after a certain amount of time has passed in order to reduce the likeliness of false alarms: As long as the main control is inactive, detector readings are ignored and thus false detections cannot result in tunnel closures. The two required ports either activate the end control or close the tunnel, whereas the `VehiclesEntering` provided port is abstract, forcing all derived types to override its behavior.

```
abstract class MainControl : Component
{
   [Subcomponent] public readonly VehicleDetector LeftDetector = /* ... */;
   [Subcomponent] public readonly VehicleDetector PositionDetector = /* ... */;
   [Subcomponent] public readonly VehicleDetector RightDetector = /* ... */;
   [Subcomponent] public readonly Timer Timer = /* ... */;

   public extern void ActivateEndControl();
   public extern void CloseTunnel();

   public abstract void VehiclesEntering(int vehicleCount);
}
```

The S# model of original the main control design is shown next. It counts the number of vehicles that are assumed to be in the area observed by the `MainControlOriginal` component. The counter's upper bound is restricted in the component's constructor to facilitate model checking: Without an upper bound, the counter can increase indefinitely, for instance due to continuous false detections of the pre control's light barrier which in turn causes the main control's `VehiclesEntering` port to be invoked repeatedly. The `VehiclesEntering` provided port is overridden to increase the counter by the number of entering vehicles. Additionally, the timer is reset every time a vehicle enters the area in order to give the newly entered vehicles enough time to pass through.

When the main control is active, i.e., its counter is not zero, and the light barrier detects a vehicle, the other two sensors are checked to determine the appropriate action. In the original design, the main control errs on the side of caution: If the light barrier detects a vehicle but neither the left nor the right overhead detectors do, it is assumed that the vehicle drives on the left lane and thus the tunnel is closed; otherwise, the end control is activated. Vehicle detections additionally decrement the counter, at least in situations where the tunnel is not closed; in all other situations, the value of the counter

is irrelevant as the model does not contain the reset procedure required to open the tunnel again. If all vehicles have left the area or the timer has elapsed, the main control is deactivated by stopping the timer or zeroing the counter regardless of the number of overheight vehicles still assumed to be present in the area. Deactivating the main control decreases the chances of false alarms while potentially making collisions more likely depending on the values chosen for the timeout. Particularly in situations where faults caused the main control to be activated in the first place and there is thus no overheight vehicle that must be observed, deactivating the main control is important to decrease the amount of false alarms.

```
class MainControlOriginal : MainControl
{
   private int _count;

   public MainControlOriginal(int vehicleCount)
   {
      Range.Restrict(_count, 0, vehicleCount, OverflowBehavior.Clamp);
   }

   public override void VehiclesEntering(int vehicleCount)
   {
      _count += vehicleCount;
      Timer.Start();
   }

   public override void Update()
   {
      Update(LeftDetector, RightDetector, PositionDetector, Timer);

      if (_count > 0 && PositionDetector.IsVehicleDetected)
      {
         if (LeftDetector.IsVehicleDetected || !RightDetector.IsVehicleDetected)
            CloseTunnel();
         else
         {
            _count--;
            ActivateEndControl();
         }
      }

      if (Timer.HasElapsed)
         _count = 0;
      else if (_count <= 0)
         Timer.Stop();
   }
}
```

The `EndControl` component shown in the following declares an overhead detector to detect high or overheight vehicles trying to enter the tunnel on the left lane. Its timer is used to deactivate the end control after some time to reduce the chance of false detections or high vehicles triggering the detector, both of which would cause a false alarm. The original design of the end control overrides the abstract `VehicleEntering` provided port such that the timer is started whenever a vehicle is detected that enters the area. It reports a collision to be potentially imminent by invoking the `CloseTunnel` required port when it is active and its sensor detects a vehicle; as there might not really be an overheight vehicle that is detected, a false alarm could occur.

```
abstract class EndControl : Component
{
   [Subcomponent] public readonly VehicleDetector LeftDetector = /* ... */;
   [Subcomponent] public readonly Timer Timer = /* ... */;

   public extern void CloseTunnel();
   public abstract void VehicleEntering();
}

class EndControlOriginal : EndControl
{
   public override void VehicleEntering() => Timer.Start();

   public override void Update()
   {
      Update(Timer, LeftDetector);

      if (Timer.IsActive && LeftDetector.IsVehicleDetected)
         CloseTunnel();
   }
}
```

The overall `HeightControl` assembles `PreControl`, `MainControl`, and `EndControl` instances as well as a `TrafficLight` instance. The subcontrollers are passed via the constructor to support multiple variants, also setting up the connections between the ports of the three subcontrollers and the traffic light in accordance with the connections shown in Figure 7.3. The subcontrollers are updated in their natural order.

```
class HeightControl : Component
{
   [Subcomponent] public readonly PreControl PreControl;
   [Subcomponent] public readonly MainControl MainControl;
   [Subcomponent] public readonly EndControl EndControl;
   [Subcomponent] public readonly TrafficLight TrafficLight = /* ... */;

   public HeightControl(PreControl pre, MainControl main, EndControl end)
   {
      PreControl = pre;
      MainControl = main;
      EndControl = end;

      Bind(nameof(pre.ActivateMainControl), nameof(main.VehiclesEntering));
      Bind(nameof(main.ActivateEndControl), nameof(end.VehicleEntering));
      Bind(nameof(end.CloseTunnel), nameof(TrafficLight.ToRed));
      Bind(nameof(main.CloseTunnel), nameof(TrafficLight.ToRed));
   }

   public override void Update()
      => Update(PreControl, MainControl, EndControl);
}
```

Instances of the `Timer` component are used by all `MainControl`- and `EndControl`-derived components. The timer's remaining time is clamped to the global `Timeout` value declared by the `Model` class introduced later on. A value of -1 indicates that the timer is inactive, whereas a timeout occurs and the timer has elapsed when the remaining time reaches zero. Stopping the timer consequently sets the remaining time to -1 while starting it sets it to the global `Timeout`. As each macro step is assumed to represent the passing of one second, the remaining time is simply decremented in the `Update` method.

```
class Timer : Component
{
   [Range(-1, Model.Timeout, OverflowBehavior.Clamp)]
   private int _remainingTime = -1;

   public bool HasElapsed => _remainingTime == 0;
   public bool IsActive => _remainingTime > 0;

   public void Start() => _remainingTime = Model.Timeout;
   public void Stop() => _remainingTime = -1;
   public override void Update() => --_remainingTime;
}
```

7.1.4 Detector Faults and Off-Nominal Vehicle Behavior

For the height control case study, sensor failures are of primary interest. It is thus necessary to add the two transient faults `Misdetection` and `FalseDetection` to vehicle detectors as shown below, thereby injecting the faults for all `VehicleDetector`-derived types. The faults' effects affect the `IsVehicleDetected` provided port, simply returning `false` for misdetections and `true` for false detections. Misdetections might, for example, be the result of very fast-driving vehicles that cannot be detected if the sensors are not checked sufficiently often. False detections can be triggered by birds flying through a light barrier, for instance. As both effects affect the same port, simultaneous activations of both faults would lead to nondeterministic behavior, needlessly slowing down model checking as the two faults cannot affect each other. Hence, `MisdetectionEffect` is assigned a higher priority to eliminate the nondeterminism; giving precedence to `FalseDetectionEffect` instead would not have made any observable difference.

```
abstract class VehicleDetector : Component
{
   // other members as above

   public readonly Fault Misdetection = new TransientFault();
   public readonly Fault FalseDetection = new TransientFault();

   [FaultEffect(Fault = nameof(Misdetection)), Priority(1)]
   public abstract class MisdetectionEffect : VehicleDetector
   {
      public override bool IsVehicleDetected => false;
   }

   [FaultEffect(Fault = nameof(FalseDetection)), Priority(0)]
   public abstract class FalseDetectionEffect : VehicleDetector
   {
      public override bool IsVehicleDetected => true;
   }
}
```

As high or overheight vehicles are not allowed to drive on left lane in the entire height control area, vehicles driving on the left lane nevertheless are modeled using faults. Similarly, the durations of the timers are chosen based on some assumptions about traffic flow such that all vehicles can usually pass the observed main and end control areas well within the time frames set by the timeouts. When vehicles take too long to pass through the height control area due to a traffic jam, for instance, the main or end controls might

be deactivated prematurely, potentially missing an overheight vehicle that attempts to enter the tunnel on the left lane. Thus, such violations of the timing assumptions are also modeled using faults. As it is generally irrelevant which overheight vehicles drive on the left lane, there is only one fault, `LeftOHV`, whose activation allows but does not force all overheight vehicles to switch lanes. For false alarms, it is important to differentiate between high and overheight vehicles on the left lane, hence there is also a `LeftHV` fault for left-driving high vehicles. By contrast, the `SlowTraffic` fault can affect all kinds of vehicles, allowing but not forcing them to drive slower than expected.

The integration of the faults' effects into the `Vehicle` component is shown below; the effects are later associated with the corresponding `Fault` instances in the `VehicleSet`'s constructor. `ChooseLane` and `ChooseSpeed` are two virtual methods that, by default, only allow a vehicle to drive on the right lane with its maximum speed; they replace the previously shown uses of the `Choose` method in `Vehicle.Update`. The `DriveLeftEffect` overrides `ChooseLane`, allowing a vehicle to nondeterministically choose the lane it drives on. Similarly, `SlowTrafficEffect` overrides `ChooseSpeed`, allowing a vehicle to nondeterministically accelerate or decelerate as it wishes. Due to the use of S#'s nondeterministic `Choose` methods in both fault effects, each `Vehicle` instance decides independently whether it is actually affected by activations of the `LeftOHV`, `LeftHV`, or `SlowTraffic` faults. If no overheight vehicle decides to switch to the left lane, for instance, the `LeftOHV` fault is not activatable as it has no effect on the model, causing the S# run time to remove the corresponding transition as discussed in Chapter 6.

```
class Vehicle : Component
{
   // other members as shown before

   protected virtual Lane ChooseLane() => Lane.Right;
   protected virtual int ChooseSpeed() => Model.MaxSpeed;

   public override void Update()
   {
      if (IsTunnelClosed)
         return;

      _speed = ChooseSpeed();
      Position += _speed;

      if (Position < Model.EndControlPosition)
         Lane = ChooseLane();
   }

   [FaultEffect]
   public class DriveLeftEffect : Vehicle
   {
      protected override Lane ChooseLane() => Choose(Lane.Right, Lane.Left);
   }

   [FaultEffect]
   public class SlowTrafficEffect : Vehicle
   {
      protected override int ChooseSpeed()
         => ChooseFromRange(Model.MinSpeed, Model.MaxSpeed);
   }
}
```

As the three faults `LeftOHV`, `LeftHV`, and `SlowTraffic` affect multiple `Vehicle` instances, they are declared in the `VehicleSet` component, using the `AddEffects` method to dynamically associate the faults with their effects on the affected `Vehicle` instances.

```
class VehicleSet : Component
{
   // other members as shown before

   public readonly Fault LeftHV = new TransientFault();
   public readonly Fault LeftOHV = new TransientFault();
   public readonly Fault SlowTraffic = new TransientFault();

   public VehicleSet(params Vehicle[] vehicles)
   {
      // as above

      LeftOHV.AddEffects<Vehicle.DriveLeftEffect>(vehicles.Where(vehicle =>
          vehicle.Kind == VehicleKind.Overhigh));
      LeftHV.AddEffects<Vehicle.DriveLeftEffect>(vehicles.Where(vehicle =>
          vehicle.Kind == VehicleKind.High));
      SlowTraffic.AddEffects<Vehicle.SlowTrafficEffect>(vehicles);
   }
}
```

7.2 Modeling the Design Variants

The design alternatives considered in the following are inspired by the suggestions made by Ortmeier et al. [158]. In total, there is one additional pre control and end control variant each as well as three design alternatives for the main control. Consequently, there are two `PreControl`-derived component types, four `MainControl` variants, and two `EndControl` types, in total resulting in sixteen possible combinations. Of these sixteen combinations, four are not analyzed in detail as their main controls ignore the improved detection capabilities of their pre controls, which makes these combinations unrealistic. The following briefly introduces the different variants listed in Figure 7.5, only showing exemplary S# models for two of them; the remaining variants integrate in a similar way into the overall model and are available online [101].

`PreControlOverheadDetectors`. The pre control can be improved by installing one overhead detector for each lane in addition to the light barrier. These improved detection capabilities are designed to avoid collisions in situations where two overheight vehicles drive through the pre control at the same time. With the original design, the main control's counter would only be increased by one and the main control might subsequently be deactivated too soon. The following S# model of the `PreControlOverheadDetectors` component declares two additional subcomponents, namely the left and right overhead detectors. As the variant is able to distinguish between one and possibly two overheight vehicles passing by, it invokes the `ActivateMainControl` required port with more precise information on the number of entering vehicles. However, due to the technical limitations of the overhead detectors that cannot distinguish between high and overheight vehicles, the improved pre control also reports two vehicles entering the main control area in situations where only one overheight vehicle passes through and a high vehicle is on the other lane.

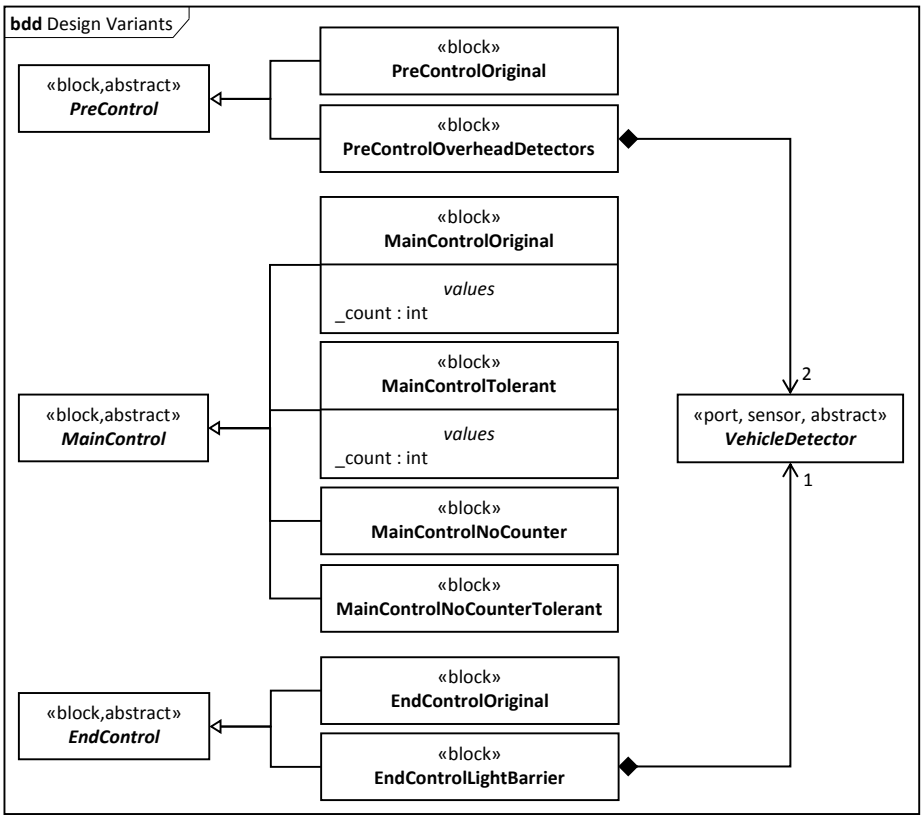


Figure 7.5. A partial block definition diagram showing all modeled variants of the PreControl, MainControl, and EndControl blocks. Some of the variants introduce new associations, while others add new values or simply declare new behavior without any structural changes. Only new associations are shown; the associations of the abstract blocks from Figure 7.1 are omitted.

```
class PreControlOverheadDetectors : PreControl
{
   [Subcomponent] public readonly VehicleDetector LeftDetector;
   [Subcomponent] public readonly VehicleDetector RightDetector;

   public override void Update()
   {
      Update(LeftDetector, RightDetector, PositionDetector);

      if (PositionDetector.IsVehicleDetected && LeftDetector.IsVehicleDetected &&
          RightDetector.IsVehicleDetected)
         ActivateMainControl(vehicleCount: 2);
      else if (PositionDetector.IsVehicleDetected)
         ActivateMainControl(vehicleCount: 1);
   }
}
```

`MainControlNoCounter`. Another possibility to avoid premature deactivations of the main control when two overheight vehicles enter its area simultaneously is to remove the counter, thus keeping the main control active until its timer elapses. Removing the counter acknowledges the fact that the main control is unable to accurately determine

the number of overheight vehicles it has to observe; the counted value can be wrong both due to fault activations as well as due to the overhead detectors' inability to distinguish between high and overheight vehicles.

`MainControlNoCounterTolerant`. The previous variant of the main control increases the probability of false alarms, as the longer the main control remains active, the more likely it is that faults or a high vehicle on the left lane trigger an alarm. This situation can be remedied by no longer erring on the side of caution: When the main control's light barrier detects a vehicle but neither of its overhead detector do, the overheight vehicle is assumed to be on the right lane. In such a situation, no alarm is raised but the end control is still activated as usual to ensure that the presumably detected vehicle does not enter the tunnel on the left lane. The `MainControlNoCounterTolerant` component shown below does not declare any new subcomponents, it merely overrides the `Update` method and the `VehiclesEntering` provided port. In case the latter is invoked, only the timer is restarted as there is no counter to update. As long as the timer is inactive or the light barrier does not detect an overheight vehicle, the `Update` method takes no further actions. Otherwise, however, it checks both overhead detectors: If neither detector reports a vehicle, the vehicle detected by the light barrier is assumed to be on the right lane and thus the end control is activated; by contrast, the original design would have the vehicle assumed to be on the left lane instead, resulting in a tunnel closure.

```
class MainControlNoCounterTolerant : MainControl
{
   public override void VehiclesEntering(int vehicleCount) => Timer.Start();

   public override void Update()
   {
      Update(LeftDetector, RightDetector, PositionDetector, Timer);

      if (!Timer.IsActive || !PositionDetector.IsVehicleDetected)
         return;

      if (LeftDetector.IsVehicleDetected && !RightDetector.IsVehicleDetected)
         CloseTunnel();
      else
         ActivateEndControl();
   }
}
```

`MainControlTolerant`. The last one of the considered design variants of the main control also tolerates high vehicles at the left overhead detector just like the previous design alternative, but in contrast to the previous variant, it still counts the overheight vehicles that it assumes to currently pass through its observed area.

`EndControlLightBarrier`. The end control is responsible for a large percentage of false alarms as it is completely unable to distinguish between high and overheight vehicles. It can thus be improved by adding a light barrier into the newly built fourth tube of the tunnel that consequently only spans the right lane to monitor overheight vehicles. Due to road layout restrictions, it is impossible to retrofit light barriers spanning only the left or both lanes which would have rendered the entire height control obsolete in the first place. Similar to the original main control variant, the improved end control counts the number of overheight vehicles it assumes to be in its observed area.

7.3 Automated Composition of the Design Variants

To instantiate a variant of the height control model, the appropriate component instances must be created, their initial states and subcomponents must be set, and their ports must be connected. Reflection is used to automatically instantiate all design variants of the model. The variants of a subcontroller of type `T` are looked up by the following `GetVariants<T>` method, returning all non-abstract subtypes of `T`. The method uses C#'s language integrated query syntax (LINQ) and .NET's reflection capabilities to check all types contained in the assembly that `T` is defined in, i.e., it searches all types contained in the compiled library that the C# compiler creates for model.

```
IEnumerable<Type> GetVariants<T>() where T : class
   => from type in typeof(T).Assembly.GetTypes()
      where type.IsSubclassOf(typeof(T)) && !type.IsAbstract
      select type;
```

Additionally, there is a helper method that is used to determine whether a selection of subtypes of `PreControl` and `MainControl` are compatible, that is, whether they form a realistic combination where the main control does not simply ignore the pre control's improved detection capabilities. Alternatively, valid model configurations could also be read from a database or be specified in a separate file, for instance. For more complex case studies and model instantiation procedures, dependency injection [171] with additional metadata about the capabilities of the individual variants could also be used in order to increase the flexibility in the specification of compatible variants.

```
bool IsRealisiticCombination(Type preControl, Type mainControl)
{
   var mainControlHasCounter = mainControl != typeof(MainControlNoCounter) &&
       mainControl != typeof(MainControlNoCounterTolerant);
   return preControl != typeof(PreControlOverheadDetectors) ||
       mainControlHasCounter;
}
```

Assuming a method `CreateVariant` that instantiates a given variant of a model as discussed later on, the following `CreateVariants` helper method can be used to instantiate all realistic combinations of design variants. Again using LINQ, the Cartesian product of all sixteen variant combinations is created, unrealistic combinations are removed using the `IsRealisiticCombination` method, and for each remaining combination, the model variant is instantiated using `CreateVariant`:

```
IEnumerable<Model> CreateVariants()
   => from preControl in GetVariants<PreControl>()
      from mainControl in GetVariants<MainControl>()
      from endControl in GetVariants<EndControl>()
      where IsRealisiticCombination(preControl, mainControl)
      select CreateVariant(preControl, mainControl, endControl);
```

To instantiate a variant, instances of the variant's subcontroller types must be created, again using .NET's reflection capabilities: Given a .NET reference type `t`, the library method `Activator.CreateInstance(t)` allocates a new object of type `t`, subsequently executing its default constructor. Using this .NET method, the `CreateVariant` method instantiates the three subcontroller types before it passes them to the constructor of the `Model` class, which in turn sets up the remaining parts of the model and connects the subcontrollers together as shown later:

```
Model CreateVariant(Type preControlType, Type mainControlType, Type endControlType)
{
   var preControl = (PreControl)Activator.CreateInstance(preControlType);
   var mainControl = (MainControl)Activator.CreateInstance(mainControlType);
   var endControl = (EndControl)Activator.CreateInstance(endControlType);

   return new Model(preControl, mainControl, endControl);
}
```

All of these helper methods are contained in the following `Model` class that represents the different variants of the case study. Like all S# models derived from `ModelBase`, the `Model` class declares the root components of the model's plants and controllers. The constructor assembles the overall model together using the subcontroller instances passed in as parameters: It first initializes a `VehicleSet` instance with two overheight `Vehicle` instances as well as one high `Vehicle`; alternatively, the number of `Vehicle` instances to create could also be specified as a constructor argument. Before a `HeightControl` instance is created, the sensors used by the three subcontrollers are connected to the `VehicleSet` by calling the `ConnectDetectors` helper method for each subcontroller. The helper method loops through all subcomponents of the given subcontroller that are of a `VehicleDetector`-derived type, connecting each detector's `ObserveVehicles` required port to the `VehicleSet`'s `ObserveVehicles` provided port. Lastly, the connection between the `VehicleSet`'s `IsTunnelClosed` required port and the `TrafficLight`'s `IsRed` provided port is established in the `Model` constructor. Once the constructor returns, all port bindings are set up and model initialization is complete.

```
class Model : ModelBase
{
   // Global constants omitted
   // Previously shown variant creation code omitted

   [Root(RootKind.Controller)]
   public HeightControl HeightControl { get; }

   [Root(RootKind.Plant)]
   public VehicleSet Vehicles { get; }

   public Model(PreControl pre, MainControl main, EndControl end)
   {
      Vehicles = new VehicleSet(new Vehicle(VehicleKind.Overhigh), new
          Vehicle(VehicleKind.Overhigh), new Vehicle(VehicleKind.High));

      ConnectDetectors(preControl);
      ConnectDetectors(mainControl);
      ConnectDetectors(endControl);

      HeightControl = new HeightControl(pre, main, end);
      Bind(nameof(Vehicles.IsTunnelClosed),
          nameof(HeightControl.TrafficLight.IsRed));
   }

   private void ConnectDetectors(Component controller)
   {
      foreach (var d in controller.GetSubcomponents().OfType<VehicleDetector>())
         Bind(nameof(d.ObserveVehicles), nameof(Vehicles.ObserveVehicles));
   }
}
```

7.4 Safety Analysis of the Design Variants

For all design variants of the case study, the S# framework is able to automatically analyze the hazards of collisions and false alarms by conducting DCCAs. The evaluation results are always reproducible as the analyses are completely deterministic, hence an analysis of standard deviation and other stochastic properties is unnecessary. As S# does not support simultaneous analyses of multiple model variants, DCCAs must be conducted independently for all design alternatives. Nevertheless, the following results demonstrate the applicability of the S# framework and the underlying formal safety analysis techniques for the analysis of safety-critical systems at design time, that is, in parallel to system development. Chapter 8 subsequently discusses safety analyses at run time, also introducing additional optimizations to increase safety analysis efficiency.

7.4.1 Collisions

Collisions obviously cannot happen as long as no overheight vehicles drive on the left lane, that is, as long as `LeftOHV` is never activated. Consequently, the `LeftOHV` fault is contained in all minimal critical fault sets computed for the collision hazard. To speed up DCCAs, S# allows the activation of such faults to be enforced during analysis so that all fault sets not containing `LeftOHV` are not analyzed at all, reducing the number of fault sets that have to be checked for criticality by at least a factor of two. The effect on the number of checks that actually have to be carried out is even larger for the case study due to the monotonicity of the criticality property: With `LeftOHV` enforced, only 325 instead of 4421 fault sets must be checked for the original design, reducing analysis times from 1325 seconds to merely 14.7 seconds, a speedup by a factor of 90x. In the following, all design variants are analyzed with this optimization to reduce overall analysis times; for false alarms, by contrast, no such optimization opportunity exists.

Table 7.1 presents an overview of the evaluation results for the hazard of collisions for all twelve analyzed case study variants. In S#, the hazard is specified with the expression

```
m.Vehicles.Any(v => v.IsAtPosition(Model.TunnelPosition) && v.Lane == Lane.Left &&
    v.Kind == VehicleKind.Overhigh)
```

for a `Model` instance `m` by making use of .NET's `Any` method over arrays. Consequently, the hazard occurs when there is an overheight `Vehicle` contained in `m` that has collided with the tunnel on the left lane. Simply judging from the amounts of minimal critical fault sets or their average cardinalities, some of the designs seem better than others when only considering the hazard of collisions, though the safety of a design is best assessed by computing hazard probabilities. Established techniques could be used that compute these probabilities from minimal critical fault sets [80], though these lie outside the focus of this thesis. For the original case study design, the following four minimal critical fault sets are found:

- **`LeftOHV`, `SlowTraffic`.** The most severe minimal critical fault set actually identifies a design flaw of the original system design, that is, the empty fault set would have been minimal critical had the `LeftOHV` and `SlowTraffic` faults not been added to the model explicitly. Two overheight vehicles might enter the main control area simultaneously on both lanes, but as the pre control cannot differentiate between one

Hazard: Collision

Pre	Main	End	# Faults	# Checked Fault Sets		# Minimal Critical Sets		Time
Org	NoCnt	LB	15	86	1543	7 (∅ 2.7)	61.3s	15.5m
Org	NoCnt	Org	13	61	644	4 (∅ 2.5)	25.1s	4.7m
Org	NoCntTol	LB	15	52	517	5 (∅ 2.0)	3.0s	45.3s
Org	NoCntTol	Org	13	50	515	3 (∅ 2.0)	11.6s	4.2m
Org	Org	LB	15	74	903	7 (∅ 2.6)	6.0s	96.6s
Org	Org	Org	13	53	325	4 (∅ 2.0)	3.4s	14.7s
Org	Tol	LB	15	44	262	6 (∅ 2.0)	3.5s	56.9s
Org	Tol	Org	13	42	260	4 (∅ 2.0)	1.7s	20.9s
Det	Org	LB	19	167	14860	12 (∅ 2.8)	55.7s	49.0m
Det	Org	Org	17	128	5640	8 (∅ 2.8)	23.2s	13.7m
Det	Tol	LB	19	86	4102	6 (∅ 2.0)	9.9s	24.8m
Det	Tol	Org	17	84	4100	4 (∅ 2.0)	3.3s	8.4m
							total: 3.5m	2.1h

Table 7.1. DCCAs for the hazard of collisions were conducted for all twelve case study designs on a 3.4 GHz quad-core CPU with the `LeftOHV` fault enforced. The design variants have different numbers and sizes of minimal critical sets; the average cardinality is also shown for each design variant. The table shows the analysis time differences between DCCAs with and without the maximum safe fault sets heuristic introduced in Section 6.4; the first numbers in the "# Checked Fault Sets" and "Time" columns correspond to DCCAs with the heuristic, the second numbers to DCCAs without it. Total analysis time is reduced from approximately 2 hours to roughly 3.5 minutes with the heuristic as it significantly reduces the number of fault sets that have to be checked for criticality.

or two vehicles passing its light barrier, the main control's counter is only increased by one. If the vehicle on the right exits the main control area before the slower-driving vehicle on the left lane, the main control is deactivated and the other vehicle can pass through undetected. Subsequently, the overheight vehicle on the left is able to pass the end control's overhead detector undetected when it takes long enough for the end control's timer to deactivate the last overhead detector. Consequently, no actual component faults have to be activated in order for the hazard to occur, only the observed vehicles have to behave "sufficiently inappropriate".

- **`LeftOHV`, `MisdetectionLB-Pre`.** A misdetection of the pre control's light barrier results in the main control and subsequently the end control never getting activated. Thus, an overlooked overheight vehicle is free to drive on the left lane, entering and colliding with the tunnel as none of the remaining sensors that is passes are active.
- **`LeftOHV`, `MisdetectionLB-Main`.** Similarly, a misdetection of the main control's light barrier results in the end control never getting activated, even when an overheight vehicle passes the light barrier on the left lane: No tunnel closure is initiated if the left overhead detector detects the vehicle but the light barrier does not, in which case the vehicle is assumed to be a regularly high one. Consequently, the overheight vehicle can continue on the left lane, pass the inactive end control, and cause a collision.

- **`LeftOHV`, `MisdetectionOD-End-Left`.** Another possibility of an overheight vehicle colliding with the tunnel requires a misdetection of the end control's overhead detector. If an overheight vehicle drives on the right lane throughout the entire main control area, the end control is activated when the overheight vehicle leaves the main control. If it subsequently switches to the left lane, the misdetection fault prevents its detection by the end control's overhead detector. Consequently, no alarm is raised and the overheight vehicle is allowed to proceed towards the tunnel.

The design flaw exposed by the DCCA for the original case study design is fixed by some design variants with improved pre control detection capabilities, for example. These variants can indeed detect two overheight vehicles entering the main control area simultaneously, thus { `LeftOHV`, `SlowTraffic` } is no longer a minimal critical fault set. Additionally, the main control variants without counters are also specifically designed to counteract this fault set; the minimal critical fault sets show that they indeed succeed in doing so, except when these variants are combined with the improved end control which reintroduces the problem as it also counts overheight vehicles.

7.4.2 False Alarms

For the hazard of collisions, it would be best to never deactivate any parts of the height control at all. The deactivations were introduced into the design in order to decrease the likelihood of false alarms, accepting a corresponding increase in the likeliness of collisions. It is therefore necessary to find an acceptable tradeoff between these two antagonistic hazards, for example using cost functions [80] or genetic algorithms that search for Pareto optimal tradeoffs using a probabilistic model checker as their fitness function [68, 187]. However, such approaches for balancing antagonistic safety goals are outside the scope of this thesis.

Table 7.2 presents an overview of the evaluation results for the hazard of false alarms for all twelve analyzed case study variants. The different designs have very similar numbers and sizes of minimal critical fault sets; in fact, there are only two different sets of minimal critical fault sets for all design variants. For false alarms, the only design changes that actually have an effect are more tolerant main controls that do not immediately close the tunnel as soon as an overheight vehicle is presumably detected on the left lane. All other variants do not influence the minimal critical fault sets at all, even though they might indeed affect the overall hazard probability by reducing the number of situations in which false alarms can occur. For example, the improved end control variant reduces the amount of time the end control is enabled, allowing less situations in which high vehicles or false detections of the end control's left overhead detector cause a false alarm; such probabilistic considerations, however, are outside the scope of this thesis. In S#, the hazard is specified with the expression

```
m.HeightControl.TrafficLight.IsRed &&
   !m.Vehicles.Any(v => v.Lane == Lane.Left && v.Kind == VehicleKind.Overhigh)
```

for a `Model` instance `m`, that is, the tunnel is closed but there is no overheight vehicle on the left lane anywhere within the entire height control area. For the original case study design, the following five minimal critical fault sets are found:

Hazard: False Alarm

Pre	Main	End	# Faults	# Checked Fault Sets		# Minimal Critical Sets		Time
Org	NoCnt	LB	15	62	1029	5 (∅ 1.0)	4.3s	2.7m
Org	NoCnt	Org	13	43	261	5 (∅ 1.0)	2.3s	24.5s
Org	NoCntTol	LB	15	98	5124	4 (∅ 1.5)	58.7s	59.0m
Org	NoCntTol	Org	13	73	1284	4 (∅ 1.5)	10.5s	3.3m
Org	Org	LB	15	62	1029	5 (∅ 1.0)	5.9s	3.9m
Org	Org	Org	13	42	261	5 (∅ 1.0)	2.9s	34.8s
Org	Tol	LB	15	98	5124	4 (∅ 1.5)	293s	3.4h
Org	Tol	Org	13	73	1284	4 (∅ 1.5)	35.3s	10.1m
Det	Org	LB	19	112	16389	5 (∅ 1.0)	11.8s	1.6h
Det	Org	Org	17	85	4101	5 (∅ 1.0)	5.4s	14.2m
Det	Tol	LB	19	160	81924	4 (∅ 1.5)	529s	3.6d
Det	Tol	Org	17	127	20484	4 (∅ 1.5)	75.1s	4.3h
							total: 17.2m	4d

Table 7.2. DCCAs for the hazard of false alarms were conducted for all twelve case study designs on a 3.4 GHz quad-core CPU. The table shows the analysis time differences between DCCAs with and without the maximum safe fault sets heuristic introduced in Section 6.4; the first numbers in the "# Checked Fault Sets" and "Time" columns correspond to DCCAs with the heuristic, the second numbers to DCCAs without it. The total analysis time is reduced from approximately 4 days to roughly 17 minutes thanks to the heuristic that significantly reduces the number of fault sets that have to be checked for criticality. The heuristic is even more effective than for the hazard of collisions because the hazard's minimal critical fault sets are smaller.

- **`LeftHV`.** A high vehicle on the left lane can cause a false alarm whenever the main control is active and an overheight vehicle passes the main control's detectors simultaneously on the right lane. Additionally, false alarms happen when the end control is active and a high vehicle tries to enter the tunnel on the left, in which case the tunnel is also closed unnecessarily. If left-driving vehicles had not been explicitly modeled with faults, the empty set of faults would be minimal critical again, showing that the overhead detectors' inability to distinguish between high and overheight vehicles is not systematically addressed by the original design. In other words, the original design is functionally incorrect with regard to both hazards.
- **`MisdetectionOD-Main-Right`.** As the original main control errs on the side of caution, a misdetection of its right overhead detector causes the tunnel to be closed when an overheight vehicle on the right lane triggers the main control's light barrier. In such a situation, the main control assumes one of its overhead detectors to have missed an overheight vehicle and closes the tunnel just in case.
- **`FalseDetectionLB-Main`.** For the same reason, a false detection of the main control's light barrier without a corresponding vehicle detection by any of its overhead detectors also causes the main control to close the tunnel as a precaution. After all, the main control again has to assume an overheight vehicle might be driving on the left lane that could have been missed because of a misdetection of its left overhead detector.

- **`FalseDetectionOD-Main-Left`.** If an overheight vehicle passes the main control's light barrier on the right lane, but a false detection of the left overhead detector causes it to spuriously report an overheight vehicle on the left lane, a false alarm results.
- **`FalseDetectionOD-End-Left`.** A false detection of the end control's overhead detector while the end control is active can immediately result in false alarms.

None of the analyzed design variants are able to prevent left-driving high vehicles from triggering false alarms. Even when the improved end control is used that is specifically designed to decrease the likelihood of false alarms, { `LeftHV` } is still a minimal critical fault set: It is always possible for a high vehicle to pass the end control's overhead detector on the left lane before an overheight vehicle passes its light barrier on the right that would have deactivated the end control. However, as the end control is activated for shorter amounts of time, overall false alarm probability decreases [158].

Summary and Outlook. The S# model of the height control case study makes use of many advanced S# modeling features introduced in Chapter 4 for reasons of conciseness and to follow the systematic modeling approach for safety-critical systems outlined in Chapter 3. Additionally, the model takes advantage of S#'s flexible model composition capabilities and .NET's support for reflection to compose different orthogonal design variants together. The formal safety analyses with DCCA are based on S#'s integration of the formal analysis techniques presented in Chapters 5 and 6. The evaluation of the twelve design variants for the hazards of collisions and false alarms shows the applicability of the S# framework and the underlying formal analysis techniques, also highlighting the effectiveness of the maximal safe fault sets heuristic introduced in Chapter 6. Compared to previous analyses by Ortmeier et al. [158], the S# model benefits from the modeling language's higher level of expressiveness, in particular for modeling and analyzing the design variants. For the original case study design, the S# framework is able to compute all minimal critical fault sets within three seconds; the previous analyses using an SMV-based model checker took a couple of minutes to find the same minimal critical fault sets, albeit on significantly slower hardware [159]. At the time of writing, however, S# is not yet able to balance the two orthogonal safety goals, that is, collision prevention and prevention of false alarms, requiring manual computations of hazard probabilities using established techniques [80, 156]. Support for probabilistic model checking is being worked on to support automated and more precise computations of hazard probabilities and to balance antagonistic safety goals [68].

Summary. Self-organizing systems present a challenge for model-based analysis techniques: At design time, their actual configurations are unknown, making it necessary to postpone the analyses to run time. At run time, however, model checking-based safety analysis techniques are often too time-consuming because of the large state spaces and the high number of faults that have to be analyzed. To make such run time analyses feasible, the self-organizing and non-self-organizing parts of the systems must be modeled and analyzed separately. Some additional heuristics and optimizations help to further decrease the number of different fault sets that DCCA has to check for criticality. Similar techniques are used to test self-organization mechanisms and adaptive robot systems.

Publications. The run time analysis approach is outlined in [81] for self-organizing systems and in [51] for adaptive robot systems. The efficiency optimizations are published in [120] and the testing approach is presented in [52].

8

Run Time Analysis of Self-Organizing Systems

Safety-critical self-organizing systems dynamically adapt their behavior and structure to changes in their environment, new or failing components, or new goals that the system should fulfill. The resulting system configurations typically cannot be predicted at design time, making it impossible to conduct complete safety analyses during system development. Additionally, high fault tolerance is one of the major advantages of many self-organizing production systems such as the robot cell and personalized medicine case studies introduced in Chapter 2 and considered in the following. Such self-organizing systems therefore have many rather large minimal critical fault sets, increasing the effort required to find them all using DCCA. Yet, establishing the exact limits of the system's fault tolerance, i.e., the boundaries within which the self-organization mechanism is capable of working around fault activations, is the primary concern for determining a

self-organizing system's safety. Consequently, safety analyses of self-organizing systems are carried out precisely to identify the circumstances in which the system is incapable of continuing to provide its functionality because of too many component faults that use up all of the system's available redundancy.

Self-organizing systems therefore poses a challenge for model-based safety analysis techniques: On the one hand, the typically unbounded number of possible system configurations makes it impossible to conduct safety analyses solely during development where enough time is likely available to find the large minimal critical fault sets. At run time, on the other hand, the exact system configurations are known, but there are significantly more stringent efficiency requirements for the analysis techniques to be useful. While the most likely system configurations can be analyzed at design time to rule out systematic design errors, there is no real alternative to postponing model-based analyses to run time when the actual system configurations are known. Section 8.1 introduces a systematic model-based safety analysis approach for self-organizing systems that is conducted at run time making use of S#'s flexible model composition capabilities. While model checking at run time, or, more generally, verification at run time are common in the area of self-organizing systems [201, 202], this chapter focuses on run time safety analyses that only play a subordinate role in prior work [48].

This chapter has two main contributions: The first one is the systematic classification of faults as either tolerable or intolerable [9], depending on whether the system's self-organization mechanism is able to compensate their activations. Based on this classification, DCCAs can be modularized by examining these two classes of faults separately. The second main contribution are two approaches for run time DCCAs discussed and evaluated in Section 8.2 that reduce analysis times: Safe fault sets heuristics for self-organizing resource-flow systems as well as a faster but less precise analysis mode using forced fault activations. Due to S#'s model execution approach, it is also possible to incorporate actual reconfiguration mechanisms of self-organizing systems into the S# models, allowing their functional correctness to be tested using DCCA as outlined in Section 8.3. Moreover, an approach for analysis and testing of adaptive robot systems is briefly outlined in Section 8.4.

8.1 Modeling and Analysis Approach for Self-Organizing Systems

Nafz et al. [146] and Seebach et al. [180] developed the restore invariant approach (RIA) to specify, model, and develop self-organizing systems. Even though the actual system behavior cannot be fully predetermined, there is a corridor of correct behavior within which the system is free to execute its tasks. RIA enables the specification of functional requirements of self-organizing systems, making it possible to cope with self-organization mechanisms and to analyze their functional correctness. By specifying a predicate over the system's states, it is possible to efficiently determine whether the current system state is valid or invalid by checking for violations of the predicate. The predicate induces a corridor of correct behavior illustrated in Figure 8.1 that the system remains in as long as the RIA predicate holds and leaves as soon as the RIA predicate is violated. Once a temporary predicate violation occurs, for instance due to hardware

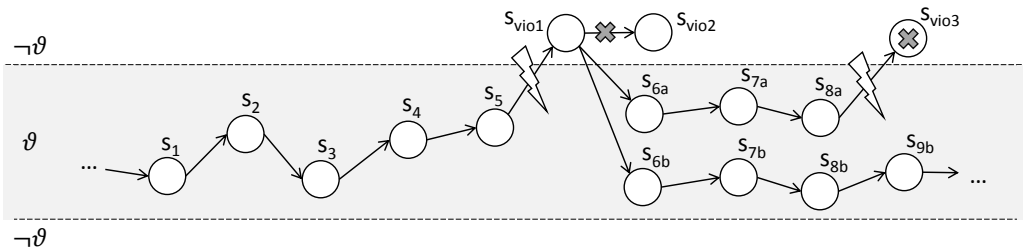


Figure 8.1. Conceptual overview of the corridor of correct behavior [146]. The corridor is formally denoted by the RIA predicate ϑ that must be satisfied during normal operation. While the system states s_1 to s_5 lie within the corridor, some event like a hardware failure causes a predicate violation and consequently a breach of the corridor by reaching invalid state s_{vio1}. The RIA approach guarantees that the system cannot reach another invalid state s_{vio2} from s_{vio1}; doing so would indicate an implementation bug. Instead, the system reconfigures itself in one of potentially many ways such that the system gets back into the corridor as indicated by states s_{6a} and s_{6b}. There are also states such as s_{vio3} that permanently violate the corridor; in such a case, the system's self-organization capabilities are exhausted and execution is halted.

failures, the self-organizing system reconfigures itself such that its behavior falls back into the corridor. As long as the system correctly detects invalid system states through a violated predicate and its reconfiguration mechanism finds at least one new, valid configuration, the system is able to return into the corridor and all of its expected functional properties continue to hold.

Since even a self-organizing system has only finite redundancy, multiple fault activations can lead to situations where the system is unable to find a new configuration. The system thus stays outside of the corridor and is no longer functioning, i.e., the self-organization mechanism's limit is reached. However, the circumstances and situations in which a self-organizing system is no longer able to find any valid configurations cannot be easily deduced from the RIA predicate. By contrast, model checking-based safety analysis techniques like DCCA are in fact able to do so if the hazard is specified as the non-existence of further valid configurations. The minimal critical fault sets determined by DCCA for this hazard therefore characterize the boundaries of the RIA corridor, allowing the evaluation of the system's overall resilience to component failures. It is thus possible to judge the effectiveness of the self-organization mechanism and the actual level of redundancy contained in the system.

A general architecture for systems implementing the restore invariant approach is illustrated by Figure 8.2. The system consists of a functional part that is for the most part unaware of all self-organization concerns while it executes the system's nominal behavior. Additionally, there is an observer/controller that observes the functional part for corridor violations. Once a RIA predicate violation is detected, a reconfiguration mechanism is triggered that tries to bring the system back into the corridor [175]; a result checker is used to verify that the new configuration is indeed valid. The algorithm used by this reconfiguration mechanism is intentionally underspecified; it could, e.g., use constraint solving or genetic algorithms [165, 178]. Due to its random nature, the latter might fail to find any valid configurations even though there actually are some.

The robot cell and the personalized medicine case studies introduced in Chapter 2 are two self-organizing resource-flow systems that make use of the restore invariant approach. They are based on the same metamodel for self-organizing resource-flow systems introduced by Nafz et al. [146] and Seebach et al. [180] that is outlined in Figure 8.3. Table 8.1 explains how the case studies map to the metamodel. There is a set of system class-specific constraints that make up the RIA predicate for self-organizing resource-flow systems [180]: For instance, the Roles allocated to an Agent may only include those Capabilities that it has available and the ports specified in the Role's pre- and postcondition require the corresponding input/output connections between the Agents. Among others, these two constraints define the corridor of correct behavior for all systems based on the metamodel. However, as the Process Capabilities of the personalized medicine case study have amounts which they can run out of despite the Capability being otherwise fully functional, some additional constraints are required for the case study in order to compensate for this minor deviation from the overall metamodel. In particular, for all recipes within the system, a Role assigned to a station is only valid when the station does not have to dispense more of any ingredient type than it has available.

8.1.1 Safety Analysis at Run Time

The S# framework can be used for safety analyses during the development of self-organizing systems as long as only a few system configurations are considered. Such design time safety analyses for a selection of likely system configurations help to identify and fix design mistakes that would result in unacceptable safety risks during system operation. More complete analyses, however, are only possible at run time when the actual system configurations are known: Instead of all possible configurations, only the current one is analyzed, requiring additional analyses whenever the configuration changes. New configurations typically result from the reconfiguration mechanism being triggered, for instance by fault activations and subsequent RIA predicate violations; alternatively, it is also possible that the system tries to find a more optimal configuration. Additionally, new tasks can enter the system at run time, e.g., new recipes must be processed in the personalized medicine case study, also triggering reconfigurations.

In the context of self-organizing systems, commissioning describes the process of bringing a new configuration into a working condition by conducting the appropriate tests and run time safety analyses. Figure 8.4 gives an overview of the four different types of commissioning [125] that are available for the S#-based analysis of self-organizing resource-flow systems. The actual plants and controllers used in reality can be combined with their S# models in four different ways depending on the intended use:

- **Constructive Commissioning.** Analyzing the S# models of the plants in conjunction with the S# models of the controllers results in classical design time analysis, for instance when analyzing a selection of system configurations during development.
- **Virtual Commissioning.** The real controller software is analyzed in conjunction with a S# model of the plants. In this scenario, the actual controller is executed while plant, sensor, and actuator behavior is abstracted by S#, allowing the effects of faults to be checked without requiring or endangering actual hardware components.

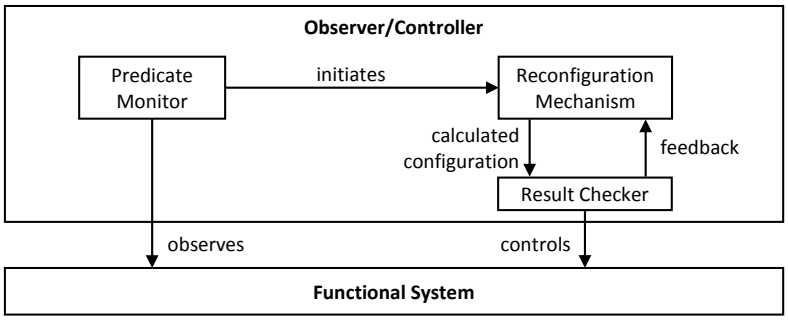


Figure 8.2. A system architecture implementing the restore invariant approach [146]. The observer/controller monitors and reconfigures the functional parts of the system whenever a corridor violation is detected; new configurations are validated by a result checker.

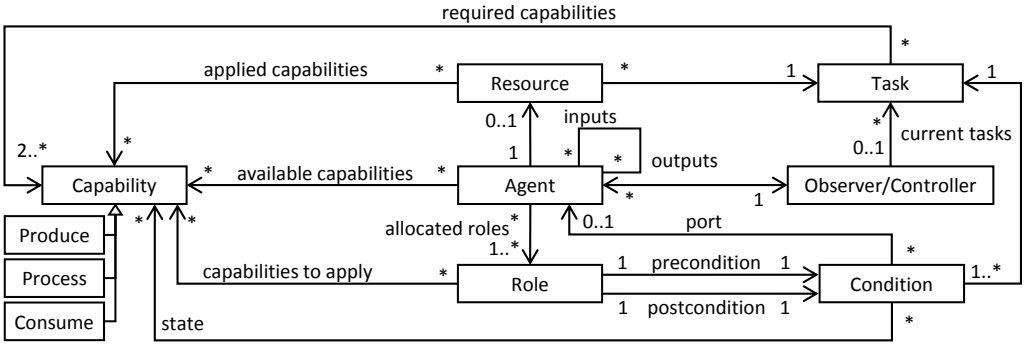


Figure 8.3. A class diagram giving a simplified overview of the metamodel for self-organizing resource-flow systems: Resources are passed along a set of Agents, each applying certain Capabilities in order to introduce a Resource into the system (Produce), to remove a Resource from the system (Consume), or to conduct a step towards the completion of the Resource's Task (Process). The Observer/Controller monitors the Agents and assigns their Roles such that all Resources are eventually fully processed with the correct order of Capability applications. Such a resource flow is specified by the pre- and post-Conditions of all Roles within the system, as well as the inputs and outputs of the Agents that establish their interconnections.

Metamodel	Robot Cell Case Study	Personalized Medicine Case Study
Resource	workpiece	pill container
Task	task	recipe
Agent	robots and carts	stations and conveyor belts
Produce	workpieces enter the system	pill containers are loaded onto a conveyor belt
Consume	workpieces leave the system	pill containers are palletized
Process	tool application	a specific amount of particulate is dispensed

Table 8.1. An overview of how the metamodel concepts shown in Figure 8.3 map to the robot cell and personalized medicine case studies. The remaining concepts not listed in the table are generic, i.e., they do not require specific mapping to the concrete case studies.

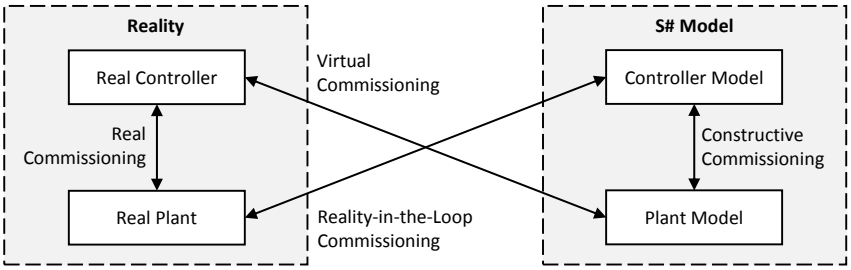


Figure 8.4. Overview of four different types of commissioning [125], combining actual system components and S# models in various ways: Real commissioning combines the actual plants and controllers to operate the system, whereas constructive commissioning is the exact opposite working on the models only. Virtual commissioning analyzes the real controllers in conjunction with the S# model of the plants; reality-in-the-loop commissioning does the opposite.

- **Reality-in-the-Loop Commissioning.** The real plants are analyzed in conjunction with a S# model of the controller, e.g., to validate the modeled controller behavior against actual plant behavior. In the following, reality-in-the-loop commissioning is used in the sense that actual system configurations are mirrored back onto the analyzed S# models for subsequent constructive or virtual commissioning at run time.
- **Real Commissioning.** The actual installation and operation of a new configuration in reality reconfigures the real plant and controller components such that actual production can continue after a preferably short testing phase.

During system operation, new configurations must be commissioned as quickly as possible in order to avoid costly downtimes. However, it cannot be ruled out that the new configurations are potentially unsafe, possibly damaging the plant equipment or causing other hazards. It is therefore often desirable to have a virtual commissioning phase where the new configuration is extensively analyzed before real commissioning is started. A tradeoff must be found between the granularity of the analyses that are carried out and the time it takes to conduct them; they cannot be too time-consuming in order to avoid prolonged downtimes, but they also cannot be too coarse-grained as disastrous safety risks might be overlooked otherwise. Section 8.2 discusses this tradeoff in further detail, presenting a way that significantly increases analysis efficiency while keeping the chances of missing minimal critical fault sets reasonably small.

Run time safety analyses with S# represent a mixture of virtual and reality-in-the-loop commissioning as illustrated by Figure 8.5. Before a new configuration is commissioned, its minimal critical fault sets are determined using the optimization techniques introduced in Section 8.2. While some of these techniques can indeed uphold DCCA's completeness guarantees, others give slightly less precise results in the sense that some minimal critical fault sets can be overlooked. It is therefore necessary to continue the analyses during and after real commissioning when more time is available for complete analyses. Additionally, the system must be continuously monitored for new conditions that result in further reconfigurations. The actual system and an instance of its S# model are therefore executed in parallel and synchronized continuously, mirroring the system's actual states in the S# model to allow for configuration-specific analyses. In

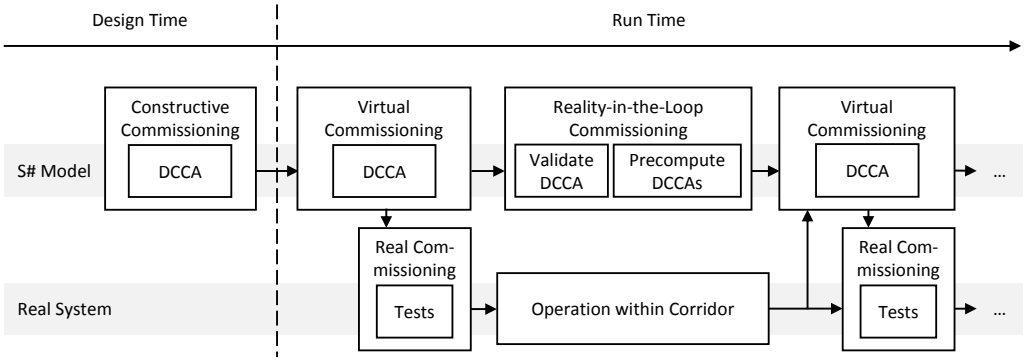


Figure 8.5. Overview of the run time safety analysis approach with S#: DCCAs are conducted at design time for a few selected system configurations in order to find critical design or implementation flaws. At run time, additional DCCAs are carried out on the actual system configurations to identify all minimal critical fault sets before the configurations are commissioned. As long as the system operates within the corridor of correct behavior, the possibly slightly imprecise DCCAs during virtual commissioning are continuously validated and additional DCCAs precompute the minimal critical fault sets for the most likely successor configurations. When the corridor is violated, the reason for the violation and the new configuration determined by the system's reconfiguration mechanism is mirrored back to the S# model, using the precomputed successor configurations to quickly assess the safety of the new configuration. If the new configuration is deemed to be safe, it is commissioned to the real system. The individual parts of this run time safety analysis approach are presented in the following, including some evaluations. However, the approach was not yet applied to a real system at the time of writing.

that sense, reality-in-the-loop commissioning is used to regularly update the model such that it accurately reflects the system's current state and configuration. Thus, only those configurations are analyzed that are actually encountered during system operation.

Additionally, it is possible to preanalyze likely successor configurations while the system still is within the corridor: Taking probabilities for the introduction of new tasks, agents, and capabilities into account as well as the activation probabilities of the faults, the most likely configurations can be guessed that the system will operate in after the next reconfiguration. Spare computational resources can be used to conduct DCCAs such that the minimal critical fault sets are already known when the next reconfiguration is required, allowing the virtual commissioning phase to be skipped or shortened.

For the robot cell case study, for example, the S#-based commissioning approach works as follows: Whenever a new task is introduced or a hardware component fails, the S# model is automatically updated to reflect these changes, for instance by activating the appropriate fault of the failing robot component. The system's self-organization mechanism computes a new configuration that is analyzed by S# before it is commissioned to the actual system. The minimal critical fault sets found by S# are evaluated to stop the commissioning process if they are deemed too risky, namely when they indicate that the system is about to reach the end of its available redundancy. Instead of waiting until the redundancy is indeed fully depleted, countermeasures can be taken to prevent future potential downtimes, for instance by replacing broken robots and tools or by adding additional robots that enlarge the smaller minimal critical fault sets.

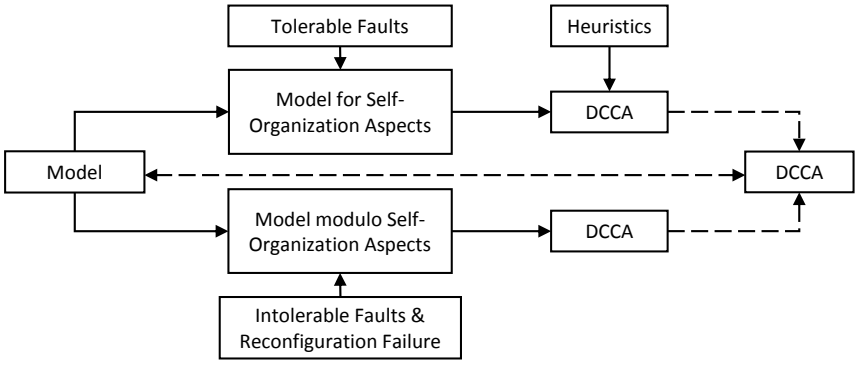


Figure 8.6. Overview of the modular safety analysis approach based in the categorization of faults into tolerable and intolerable ones: For each system configuration that is of interest, the system model is instantiated twice such that the first one includes only tolerable faults and contains all aspects that are important for the description of the self-organization mechanism. The second model includes only intolerable faults and abstracts from self-organization, in lieu thereof introducing a fault that represents failed or impossible reconfigurations. For both models, a DCCA is conducted individually. If the model modulo self-organization aspects considers all possible configurations, their combined results are equivalent to a DCCA conducted on the overall model. To speed up analyses, the heuristics introduced in Section 8.2 can be used.

8.1.2 Separation of Self-Organization Aspects

Safety analyses are often decomposed such that off-the-shelf components are not analyzed in detail as the respective vendors already determined their hazards using some safety analysis techniques. The identified hazards take on the role of component faults in the safety analyses of the systems that integrate these components [9, 186]; this is a time-tested practice with a long tradition in Fault Tree Analysis [193], for instance. Model-based safety analysis techniques can make use of the same principles to reduce the size of the analyzed models, focusing on the safety issues that result from component composition rather than the details of the individual component behavior.

This decomposition approach can be extended to self-organizing systems to modularize both the models as well as DCCAs as illustrated by Figure 8.6: Faults are manually separated into tolerable and intolerable ones [9], depending on whether the system is in principle able to tolerate their activations due to self-organization. Tolerable faults can be detected by the observer and compensated by the reconfiguration mechanism as long as a sufficient amount of redundancy is left, continuing safe operation after a reconfiguration. Consequently, the classification of faults can typically be based on the observer's ability to detect their effects. However, self-organization cannot cope with all faults: Intolerable faults are not within its reach, either because they cannot be detected or there is no appropriate reaction to the errors they cause. Additionally, the system eventually runs out of redundancy after sufficiently many tolerable faults have been activated, causing a reconfiguration failure that in turn can be seen as an intolerable fault at a higher level of abstraction. Additionally, a reconfiguration failure can also be caused by an incorrect or incomplete reconfiguration mechanism; e.g., a genetic algorithm cannot guarantee to find a valid configuration even if one still exists [165].

Consequently, S# models of self-organizing systems not only contain the relevant faults for the hazards to be analyzed, they also classify all of those faults into the tolerable and intolerable categories. Self-organization can thus be seen as a fault tolerance mechanism that tries to avoid failures by error detection and recovery [9]. In conjunction with virtual commissioning, the decomposition approach facilitates model checking-based safety analyses at run time by reducing analysis complexity: When conducting DCCAs for either tolerable or intolerable faults, virtual commissioning only considers the system's current configuration instead of all valid ones; since the number of valid configurations might be unbounded as in the case of self-organizing resource-flow systems, DCCAs would otherwise be impossible. On the other hand, the separation of tolerable and intolerable faults reduces the number of criticality checks significantly: For a model containing n tolerable faults and m intolerable ones, modularized safety analyses only have to consider $2^n + 2^m$ fault sets in the worst case. Combined DCCAs for both kinds of faults, on the other hand, would have to check 2^{n+m} fault sets for criticality.

As indicated by the dashed lines in Figure 8.6, the two individual DCCAs on the two different models containing either tolerable or intolerable faults can be equivalent to a complete DCCA conducted on the non-modularized model containing all faults. However, this correlation only holds if the model modulo the self-organization aspects would consider all valid configurations. For example, the intolerable fault of some robot R_n where it inadvertently selects the wrong tool is not critical if R_n is not part of the resource flow; as the robot is never used, none of its faults can be activated. When the system reconfigures itself to actually include R_n in the resource flow, the fault becomes activatable. Consequently, if only a single configuration is considered, the tool selection fault of R_n might be deemed safe, whereas it might actually be critical if all configurations were analyzed. Due to virtual commissioning, however, it is neither necessary nor beneficial to consider all configurations: Analysis times should be as low as possible and once the system configuration changes, new DCCAs are conducted anyway. Consequently, if the tool selection fault of robot R_n was not determined to be critical for the previous configuration where R_n was inactive, it is correctly identified as critical as soon as the system enters a configuration in which R_n is indeed active.

8.1.3 Characterization of the Corridor of Correct Behavior

DCCAs for the tolerable faults of a self-organizing resource-flow system determine the limits of its self-organization mechanism when checking for the hazard of a reconfiguration no longer being possible. For a minimal critical fault set Γ computed by such a DCCA, activations of all of the contained faults $f \in \Gamma$ prevent the system from further self-organization, causing an irrevocable violation of the corridor of correct behavior. Thus, DCCAs are able to precisely characterize the boundaries of the corridor of correct behavior. Each additionally activated tolerable fault can be seen as narrowing the corridor, leaving the system fewer possibilities to reorganize itself until eventually, a minimal critical fault set is completely activated and no further configurations exist. On the other hand, introducing additional redundancy into the system widens the corridor again, increasing the number of valid configurations that the system can continue to function in. The minimal critical fault sets have to be recomputed after each reconfiguration of

the system, as they can change in arbitrary ways after a reconfiguration: For example, when robots are removed from the robot cell or tools break, some minimal critical fault sets become smaller. On the other hand, newly installed robots introduce new minimal critical fault sets or make previously existing ones larger.

Formally, a self-organizing system is assumed to be given as a fault-aware Kripke structure K with a proposition $\varrho \in P(K)$ that is set by the reconfiguration mechanism when it can no longer find a valid configuration. Additionally, the RIA predicate $\vartheta \in P(K)$ is also assumed to be explicit in the Kripke structure, with a state $s \in S(K)$ labeled with ϑ if the predicate holds in s, that is, the state is within the corridor. A DCCA of the tolerable faults for hazard ϱ then guarantees that the predicate ϑ is not permanently violated as long as no minimal critical fault set is completely activated. In order to prove this claim, the Kripke structure K must be assumed to correctly reconfigure itself whenever the RIA predicate ϑ is violated, that is, the violation must be detected and either subsequently fixed or no reconfiguration is possible and the ϱ flag is set. This notion of correctness specifically does not require the reconfiguration mechanism to find a configuration if there still is one, thus it is also compatible with self-organizing systems based on imperfect genetic algorithms. In other words, the boundaries of the corridor of correct behavior are characterized relative to the strength of the reconfiguration mechanism. Functional correctness is reasonable to assume as Nafz et al. [146] formally verified the correctness of the result checker for self-organizing resource-flow systems that ensures the validity of the configurations computed by the reconfiguration algorithm. Thus, the reconfiguration mechanism either computes a valid configuration or none at all; the latter case is formally represented by the ϱ flag being set in those states without any further configurations. Additionally, Section 8.3 introduces a testing approach for the functional correctness of self-organization mechanisms.

Definition 8.1 (Correct Reconfiguration)**.** *For a fault-aware Kripke structure K, a RIA predicate $\vartheta \in P(K)$, and a flag $\varrho \in P(K)$ that marks states in which no reconfiguration is possible, K* correctly reconfigures itself *if $K \models \mathbf{G}(\neg\vartheta \rightarrow (\mathbf{F}\,\vartheta \vee \mathbf{F}\,\varrho))$.*

Proposition 8.1. *For a fault-aware Kripke structure K that correctly reconfigures itself, a RIA predicate $\vartheta \in P(K)$, a flag $\varrho \in P(K)$ that marks states in which no reconfiguration is possible, as well as the set Λ_ϱ of all minimal critical fault sets for hazard ϱ, $K \models (\bigwedge_{\Gamma\in\Lambda_\varrho} \neg \bigwedge_{f\in\Gamma} \mathbf{F}\,f) \rightarrow \mathbf{G}\,\mathbf{F}\,\vartheta$.*

Proof. Assume for a contradiction that there is a $\varsigma \in \mathit{paths}(K)$ such that $\varsigma \not\models \mathbf{G}\,\mathbf{F}\,\vartheta$ despite $\varsigma \models \bigwedge_{\Gamma\in\Delta} \neg \bigwedge_{f\in\Gamma} \mathbf{F}\,f$. Let $k_1, k_2, \ldots$ be the indices of those states with $\varsigma, k_i \models \neg\vartheta$ for all i. If there is no such k_i, we have a contradiction to the assumption that $\varsigma \not\models \mathbf{G}\,\mathbf{F}\,\vartheta$. Otherwise for all $i, \varsigma, k_i \models \mathbf{F}\,\vartheta \vee \mathbf{F}\,\varrho$ due to the assumed reconfiguration correctness. Due to Theorem 5.3, completeness of DCCA, there is no i such that $\varsigma, k_i \models \mathbf{F}\,\varrho$ because $\varsigma \models \mathbf{G}\,\neg\varrho$. Consequently, $\varsigma, k_i \models \mathbf{F}\,\vartheta$ for all i, a contradiction to the assumption that $\varsigma \not\models \mathbf{G}\,\mathbf{F}\,\vartheta$. □

Proposition 8.1 thus formally establishes the relationship between the corridor of correct behavior and the minimal critical fault sets for the hazard of non-existence of further configurations: A self-organizing system based on RIA never permanently leaves the corridor as long as none of the minimal critical fault sets are fully activated.

8.1.4 Analyzing Overall System Safety

To assess the overall safety of a self-organizing system, all minimal critical fault sets consisting of system's intolerable faults have to be determined. As illustrated by Figure 8.6, a general reconfiguration failure must be introduced that subsumes all tolerable faults; additionally, it also accounts for the fact that incorrect or incomplete reconfiguration mechanisms might be unable to find valid configurations even though some still exist. Due to this subsumption of all tolerable faults by a single intolerable one, the minimal critical fault sets become significantly smaller and can be found with less criticality checks; they contain at most the reconfiguration failure, but no tolerable faults anymore. If a minimal critical fault set $\Gamma \in \Lambda_H$ for the analyzed system-level hazard H contains the reconfiguration failure f_r, it actually characterizes a set of minimal critical fault sets: For each minimal critical fault set $\Gamma' \in \Lambda_\varrho$ determined by DCCA for the tolerable faults, the reconfiguration failure in Γ is replaced by Γ', i.e., $\Lambda'_H = \{\Gamma \in \Lambda_H \mid f_r \notin \Gamma\} \cup \{(\Gamma \cup \Gamma') \setminus \{f_r\} \mid \Gamma \in \Lambda_H \wedge f_r \in \Gamma \wedge \Gamma' \in \Lambda_\varrho\}$.

However, this substitution of minimal critical fault sets is only adequate if the underlying resource-flow system is deadlock-free and fair [185]. In particular, each agent must select its roles in a fair way and there must be a coordination mechanism that prevents deadlocks. In the robot cell case study, for example, a cart C_n is not allowed to transport a workpiece to robot R_m at the same time that R_m has to place another workpiece on C_n. In the S# model of the case study, fairness is guaranteed by selecting roles in a round-robin fashion. For deadlock prevention, a very simplistic scheduling mechanism is contained in the model even though more sophisticated ones could be incorporated as well: Only one workpiece can be processed simultaneously. While the mechanism results in suboptimal system performance, it has no impact on the adequacy of the substitution. In general, however, it can influence other parts of the DCCA results that are not related to the substitution, depending on the kind of intolerable faults that are modeled and analyzed. The analyses carried out in Section 8.1.5 are not influenced by this single workpiece restriction, however.

It is important to check for minimal critical fault sets that do not contain the reconfiguration failure. If some exist, the fault tolerance that the self-organization mechanism is intended to provide can be bypassed by activations of intolerable faults. As for non-self-organizing systems, such sets hint at potential improvements that can be made to the system's design in order to improve the system's overall safety. Of particular interest are additional error discovery mechanisms [9] that are able to broaden the reach of the self-organization mechanism, thereby turning some of the intolerable faults into tolerable ones. For self-organizing resource-flow systems based on the metamodel, for instance, the reconfiguration failure is irrelevant for the hazard of damaged resources, i.e., resources to which capabilities were applied in the wrong order: Production is stopped when no valid configuration is found and hence no further incorrect processing can occur. Consequently, the self-organization mechanisms of these systems are not designed to reduce the likelihood of damaged resources; otherwise, the reconfiguration failure would be contained in some or all minimal critical fault sets for the hazard. Additional detection capabilities that trigger reconfigurations before damaging a workpiece would therefore improve overall system safety.

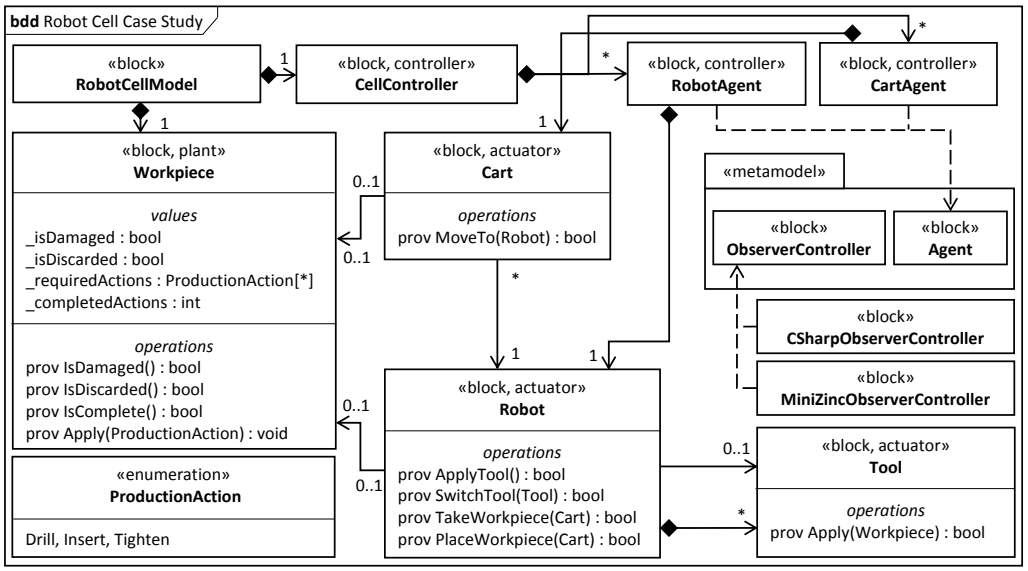


Figure 8.7. A block definition diagram giving a partial overview of the robot cell case study model. The classes of the metamodel are not shown in full detail as they can be modeled directly in S# and SysML without any modifications. The RobotAgent and CartAgent blocks instantiate the Agent class from the metamodel; they contain their respective Robot and Cart actuators, i.e., the actual hardware components that manipulate the Workpieces. There are two different implementations of the Observer/Controller metamodel class that can be used in the S# model: The MiniZincObserverController uses the MiniZinc constraint solver [148] to compute new configurations, whereas the CSharpObserverController is implemented entirely in C#.

8.1.5 Application to the Robot Cell Case Study

The S# model of the robot cell case study implements the metamodel for self-organizing resource-flow systems. All classes and associations contained in the original class diagram are directly represented in the S# model using C#'s object-oriented language features. The state machines presented by Seebach et al. [180] that describe the behaviors of the individual agents are implemented using S#'s state machine extensions for C#. In particular, the state machines control the handshake protocol between the agents that allows them to exchange resources, thereby establishing the resource flow between the robots and the carts. The RIA predicate, on the other hand, is specified as a set of lambda functions that determine violations of the corridor of correct behavior.

The S# model of the robot cell case study shown in Figure 8.7 abstracts from the distribution of the agents across different processes to simplify the model. Additionally, the position of the carts is only abstractly represented through the association between the Cart and Robot blocks, that is, a cart is conceptually located at the associated robot. Carts move instantly between their connected robots, taking no time for maneuvering. Robots and carts are controlled by their associated RobotAgent and CartAgent instances which instantiate the metamodel's Agent class. The latter represent software components that control the actual Robot and Cart hardware components; the Observer/Controller considers the software agents only and does not care about the hardware components.

The software components control their hardware counterparts through the various provided ports shown in Figure 8.7. For example, the MoveTo port instructs a cart to move to a robot, where the robot can be instructed to take or place a workpiece from or on a cart using the TakeWorkpiece and PlaceWorkpiece provided ports, respectively. The SwitchTool provided port causes a robot to switch its active tool; the model assumes tool switching to be instantaneous. Tool applications are carried out using the ApplyTool provided port that instantly applies the currently active tool to the workpiece located at the robot. Most of the aforementioned provided ports return a Boolean value that indicates whether the operation was carried out successfully. These return values abstractly model the system's error detection mechanisms, in response to which available capabilities are removed from the agents or their input/output relationships are changed. The resulting violation of the RIA predicate then causes the model to reconfigure itself.

Workpieces track the ProductionActions that have been carried out on them in order to determine whether they have been damaged during production or whether their production is complete. To do so, their _requiredActions field stores the number and order of actions that have to be conducted with the _completedActions field storing the number of successfully completed actions. If _completedActions matches the number of _requiredActions, processing is complete. The Resource class contained in the metamodel is a software representation of the actual workpieces. There is no direct link between the two; if the system is functionally correct and no faults are activated, the processing state stored in a workpiece is always consistent with the processing state stored in the corresponding resource. A workpiece is discarded if it is not fully processed when a reconfiguration takes place; the model does not contain a mechanism to continue processing a workpiece after the cell's configuration has changed.

The model makes no assumptions about the specifics of the actual observer/controller that is used. Consequently, the two specializations contained in the model, MiniZincObserverController and CSharpObserverController, can be used interchangeably. However, the original version of the former contained several implementation bugs identified by the testing approach discussed in Section 8.3 which sometimes caused it to generate invalid configurations or to overlook the existence of valid ones.

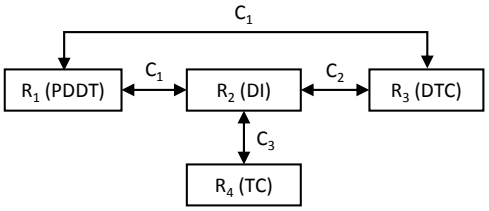
Analyzing Tolerable Faults. For the case study, the tolerable faults consist of broken tools that can no longer be applied, written as $\neg \mathsf{R}_n^{\mathsf{c}_m}$ to denote that the n-th robot's m-th capability c is broken with $\mathsf{c} \in \{\mathsf{P}, \mathsf{D}, \mathsf{I}, \mathsf{T}, \mathsf{C}\}$ for the produce, drill, insert, tighten, and consume capabilities, respectively. Additionally, there is the fault $\neg \mathsf{R}_n$ that indicates that the n-th robot is no longer able to function at all. Broken carts that can no longer move around at all are denoted by $\neg \mathsf{C}_n$ whereas a specific route between robots R_k and R_l being blocked is written as $\neg \mathsf{C}_n^{\mathsf{R}_k \leftrightarrow \mathsf{R}_l}$. For the small cell configuration shown in Figure 8.8a, five out of the 42 minimal critical fault sets are listed in Figure 8.8b: Set (1) represents the complete failure of the first cart; in that case, there are no further connections from the first robot to any others. As only the first robot has the produce capability and cannot complete the task on its own, reconfiguration fails. In the case of set (2), the second robot's insert capability breaks down which cannot be compensated by any other robot. Minimal critical fault set (3) shows that the loss of the fourth cart's route between the second and fourth robot is critical if at the same time neither the

first nor the third robot can compensate the loss of the fourth robot's tighten capability. A similar minimal critical fault set exists where $\neg \mathsf{C}_3^{\mathsf{R}_2 \leftrightarrow \mathsf{R}_4}$ is replaced by $\neg \mathsf{C}_3$. Set (4) is trivial as it represents the loss of all four drill capabilities. From set (5) follows that it is critical if only the third robot is still able to drill, but it can no longer be reached when both routes to it are blocked. In general, there are minimal critical fault sets that are trivial to deduce such as set (4); however, as soon as cart routing must be taken into consideration, the minimal critical fault sets become less obvious, especially when significantly larger configurations are analyzed. Table 8.2 gives an overview of the analysis results for configurations of various sizes; the system configuration shown in Figure 8.8 corresponds to C_1 in Table 8.2.

Analyzing Intolerable Faults. Table 8.3 shows the analysis results for the intolerable faults and the two system-level hazards of damaged or incompletely processed workpieces. Intolerable faults that are analyzed concern the workpieces that can be positioned incorrectly in front of a robot such that they are damaged during processing, or the processing task fails on them without the corresponding robot taking notice. For each processed workpiece, these two faults are individually minimal critical for the damage hazard as all of their activations either immediately lead to damage or do so when the next tool is applied. Similarly, a robot can unknowingly select the wrong tool, damaging all workpieces that it works on. However, reconfiguration failures cannot cause any damage as they stop the entire production process. Consequently, processing remains incomplete whenever reconfiguration fails, either due to an incorrect reconfiguration mechanism or when one of the minimal critical fault sets identified by the DCCAs listed in Table 8.2 is activated. Moreover, a robot or cart not receiving its updated roles after a reconfiguration is immediately minimal critical for the incomplete processing hazard, at least as long as the corresponding agent is actually required to process the workpieces in the analyzed system configuration. For inactive agents, the fault is deemed safe, but as soon as the system enters a configuration in which such an inactive agent is in fact part of the resource flow, the virtual commissioning approach correctly classifies the corresponding fault as critical. Overall, the robot cell case study only has singleton minimal critical sets of intolerable faults.

8.2 Optimizing Run Time Safety Analysis Efficiency

There are two sources of exponential complexity when analyzing tolerable faults for the hazard of non-existence of further configurations: All combinations of tolerable faults have to be checked for criticality and the model checker has to enumerate all possible combinations of tolerable fault activations during each check. DCCA's fault removal optimization helps to alleviate the second problem by removing all faults from the model that are suppressed by the analyzed DCCA formula anyway. However, the optimization is often less effective than for non-self-organizing systems due to the larger cardinalities of the fault sets that have to be checked as not as many faults can be removed from the analyzed models. Consequently, additional optimizations are necessary to cope with both the large amounts of faults as well as the large cardinalities of the minimal critical fault sets. To counter the first problem, S#'s DCCA heuristics mechanism explained in Section 6.4 is used with the two heuristics for self-organizing resource-flow systems



(a) The robot cell configuration consists of four robots with varying capabilities and three carts. C_1 has two routes, connecting R_1, R_2, and R_3.

(1) $\neg C_1$

(2) $\neg R_2^{I_1}$

(3) $\neg C_3^{R_2 \leftrightarrow R_4}$, $\neg R_1^{T_1}$, $\neg R_3^{T_1}$

(4) $\neg R_1^{D_1}$, $\neg R_1^{D_2}$, $\neg R_2^{D_1}$, $\neg R_3^{D_1}$

(5) $\neg C_1^{R_1 \leftrightarrow R_3}$, $\neg C_2^{R_2 \leftrightarrow R_3}$, $\neg R_1^{D_1}$, $\neg R_1^{D_2}$, $\neg R_2^{D_1}$

(b) An incomplete listing of five minimal critical fault sets of tolerable faults for the robot cell configuration shown on the left.

Figure 8.8. An illustration of a simple configuration of the robot cell case study showing some of its minimal critical fault sets returned by a DCCA for the hazard of non-existence of further configurations. Only tolerable faults are considered by the analysis.

	# Robots	# Carts	∅ Capabilities	∅ Routes	# Tolerable Faults	# Minimal Critical Sets	Time
$\mathcal{C}_1$	4	3	2.8	1.3	19	42 (∅ 3.0)	10.1s
$\mathcal{C}_2$	3	2	3.7	2.0	19	40 (∅ 3.0)	44.4s
$\mathcal{C}_3$	4	3	2.8	2.0	23	36 (∅ 3.0)	2.7m
$\mathcal{C}_4$	5	2	3.6	2.0	23	70 (∅ 4.0)	3.9m
$\mathcal{C}_5$	4	3	3.8	1.3	22	70 (∅ 3.0)	4.0m
$\mathcal{C}_6$	4	4	2.8	3.5	28	141 (∅ 6.0)	4.8h

Table 8.2. Overview of the evaluation results for the hazard of non-existence of further configurations with only tolerable faults considered. The system configurations are manually chosen and automatically instantiated and evaluated, differing in the number of robots and carts as well as the average numbers of available capabilities and routes between robots. Configuration $\mathcal{C}_6$ has a large amount of minimal critical fault sets with a maximum cardinality of eleven faults; more than ten million fault sets have to be checked to find them all.

		Damaged Workpiece		Incomplete Processing	
	# Intolerable Faults	# Minimal Critical Sets	Time	# Minimal Critical Sets	Time
$\mathcal{C}_1$	16	6	1.5s	7	2.3s
$\mathcal{C}_2$	19	12	1.0s	11	8.1s
$\mathcal{C}_3$	22	12	2.6s	11	42.7s
$\mathcal{C}_4$	17	6	2.6s	8	2.9s
$\mathcal{C}_5$	16	6	1.6s	8	1.9s
$\mathcal{C}_6$	17	6	2.0s	9	2.0s

Table 8.3. Overview of the evaluation results for the two hazards of damaged or incompletely processed workpieces with regard to intolerable faults. The analyzed configurations correspond to the ones in Table 8.2. All minimal critical fault sets have a cardinality of one; the analyses are aborted after all fault sets of cardinality two are checked for criticality. Both hazards can be analyzed in roughly two seconds for configuration $\mathcal{C}_6$ when only intolerable faults are considered. By contrast, a non-modularized DCCA would additionally have to consider the 28 tolerable faults from Table 8.2 that on their own already take almost five hours to analyze.

introduced in Sections 8.2.1 and 8.2.2. To cope with the second problem, fault activations are enforced during the analyses as explained in Section 8.2.3, always activating all activatable faults instead of the usual nondeterministic choice. While the heuristics affect neither soundness nor completeness of DCCA, the second optimization can in fact cause some minimal critical fault sets to be overlooked.

Safe fault sets heuristics are even more important for self-organizing systems than for non-self-organizing ones: There typically are more faults and thus more combinations of faults that have to be checked for criticality, and the higher cardinalities of the minimal critical fault sets mean that many safe fault sets are checked before any critical ones are encountered. The two heuristics introduced in the following are highly effective as evaluated in Section 8.2.4: Self-organizing resource-flow systems have many instances of the same component types, e.g., there are multiple stations and conveyor belts in the personalized medicine case study. This uniformity can be taken advantage of when conducting DCCAs by providing heuristics that exploit the similarities between the different agent instances in order to suggest large safe fault sets.

8.2.1 Fault Subsumption Heuristic

Agents often have severe faults that subsume less severe ones. For example, an activation of $\neg \mathsf{R}_n$ deactivates a robot in the robot cell case study entirely, hence any $\neg \mathsf{R}_n^{\mathsf{c}m}$ that disable R_n's m-th tool for the capability c are no longer activatable. Similarly, malfunctions of individual ingredient dispensers in the personalized medicine case study do not matter after a complete failure of the corresponding station. Consequently, the overall agent fault subsumes the individual capability faults. However, the faults still have to be modeled separately, as the subsuming faults often have additional effects, i.e., they can be more severe than the combination of all subsumed faults. For example, a station that failed completely also no longer allows any pill containers to pass through, which would still be possible if only all of the station's ingredient dispensers had failed.

Such subsumption relationships between faults are in general not statically deducible from the S# models, that is, they cannot be inferred automatically without checking all fault sets first. However, if the faults contained in a model are explicitly annotated with subsumption information, transitive subsumptions can be taken advantage of by a fault subsumption heuristic. For the aforementioned station faults, for example, fault subsumption is annotated in the S# model as follows during model initialization:

```
CompleteStationFailure.Subsumes(IngredientTanks.Select(tank => tank.TankDefect));
```

All particulate dispenser stations have a `CompleteStationFailure` fault and several ingredient tanks with a `TankDefect` fault for each ingredient type that they are able to administer into a pill container. In the example above, the `CompleteStationFailure` is declared to subsume all of the `TankDefect` faults declared by the ingredient tanks; the declaration uses .NET's `Select` method that maps each tank to its `TankDefect` fault. Formally, the subsumption relation is defined as follows:

Definition 8.2 (Fault Subsumption)**.** *For all faults $f_1, f_2 \in F(K)$ of a fault-aware Kripke structure K and a hazard H, f_1* subsumes *f_2, written as $f_1 \preceq f_2$, if and only if for all $\Gamma \subseteq F(K)$ with $f_1 \in \Gamma$, Γ is safe for hazard H implies that $\Gamma \cup \{f_2\}$ is also safe*

for H. The set of all faults subsumed by $\Gamma \subseteq F(K)$ is denoted by $\mathcal{S}(\Gamma) = \{f \mid f \in F(K)$ and $f' \preceq f$ for some $f' \in \Gamma\}$.

The subsumption relation $\preceq$ is both reflexive and transitive, and $\mathcal{S}(\Gamma)$ always is a superset of Γ due to the reflexivity:

Lemma 8.1. *The fault subsumption relationship $\preceq$ is a preorder.*

Proof. Reflexivity: Let $\Gamma \subseteq F(K)$ and $f \in \Gamma$ such that Γ is safe for a hazard H. Consequently, $\Gamma \cup \{f\} = \Gamma$ and thus $\Gamma \cup \{f\}$ is also safe for H, hence $f \preceq f$.

Transitivity: Let $f_1, f_2, f_3 \in F(K)$ such that $f_1 \preceq f_2$ and $f_2 \preceq f_3$. For a $\Gamma \subseteq F(K)$ with $f_1 \in \Gamma$ and Γ is safe for hazard H, $\Gamma \cup \{f_2\}$ is also safe for H because $f_1 \preceq f_2$. Since $f_2 \preceq f_3$, $\Gamma \cup \{f_2, f_3\}$ is also safe for H. As $\Gamma \cup \{f_3\} \subseteq \Gamma \cup \{f_2, f_3\}$, it follows that $\Gamma \cup \{f_3\}$ is also safe for H due to the monotonicity established by Theorem 5.2. □

Theorem 8.1. *For a fault set $\Gamma \subseteq F(K)$ of a fault-aware Kripke structure K and a hazard H, Γ is safe for H if and only if $\mathcal{S}(\Gamma)$ is safe for H.*

Proof. "⇒": As $F(K)$ is finite, let $\mathcal{S}(\Gamma) = \{f_1, f_2, \ldots, f_k\}$ and $\Gamma_i = \Gamma \cup \{f_j \mid 0 < j \leq i\}$ for all $i = 0 \ldots k$. Trivially, Γ_0 is safe for H by assumption. If Γ_i is safe for H, choose $f \in \Gamma \subseteq \Gamma_i$ such that $f \preceq f_{i+1}$. Consequently, $\Gamma_i \cup \{f_{i+1}\}$ is safe for H. As $\Gamma_{i+1} = \Gamma_i \cup \{f_{i+1}\}$, it follows by induction that Γ_k is safe for H and $\Gamma_k = \mathcal{S}(\Gamma)$.

"⇐": If $\mathcal{S}(\Gamma)$ is safe for H, the fact that Γ is also safe for H follows from the monotonicity established by Theorem 5.2 and the fact that $\Gamma \subseteq \mathcal{S}(\Gamma)$ due to the reflexivity of $\preceq$. □

When conducting a DCCA with the fault subsumption heuristic, S# enlarges a fault set Γ with all subsumed faults, checking whether the enlarged set is safe. If it is, Γ and all other subsets of the enlarged fault set are also known to be safe, thus reducing the number of fault sets that would unsuccessfully be checked for criticality otherwise.

Heuristic 2 (Fault Subsumption Heuristic)**.** *Before checking a fault set $\Gamma \subseteq F(K)$ for criticality for a hazard H, first check whether $\mathcal{S}(\Gamma)$ is safe for H.*

If an incorrect subsumption relation is specified in the S# model, the heuristic is still correct but less effective: Only the "downwards" direction of Theorem 8.1 is taken advantage of during analyses as the monotonicity of safe fault sets holds regardless of the validity of the specified subsumption relations. To enable the fault subsumption heuristic when conducting a DCCA, an instance of the `SubsumptionHeuristic` class for a model instance `model` must be added to the list of active heuristics:

```
var analysis = new SafetyAnalysis();
analysis.Heuristics.Add(new SubsumptionHeuristic(model));
```

8.2.2 Minimal Redundancy Heuristic

All analyzed fault sets are safe as long as the system still has redundancy reserves that it can make use of. For self-organizing resource-flow systems, there are several levels of redundancy: For example, there are multiple tool instances for the same capabilities to compensate for broken tools; additionally, there are multiple routes established by the carts or conveyor belts to transport resources between the agents; and lastly, there are multiple agents that replicate the routes and capabilities of other agents in order to replace them when they fail completely.

The minimal redundancy heuristic takes these levels of redundancy into account by systematically shrinking the amount of left-over redundancy. For example, if there are three stations that are fully connected with each other and all of them have particulate p available, each individual dispensing failure is not critical. The heuristic therefore suggests to check all fault sets that contain two dispensing failures for p. The quality of the suggestions made by the heuristic depends on the connections available between the agents; in the aforementioned example, most of the suggestions would be critical if only one of the stations is actually able to administer p due to resource-flow restrictions. For very sparsely connected configurations in which there are only very few possible resource flows, the heuristic is therefore expected to be at least ineffective if not outright counterproductive as it suggests many fault sets that turn out to be critical. In between the best and the worst case, a top-down search strategy can be used for suggested sets that turn out to be critical: By analyzing all subsets of such inappropriately suggested fault sets by decreasing instead of the usual increasing level of cardinality, maximal safe subsets are likely identified faster.

Let $C_1, \ldots, C_n$ be sets of capability instances contained in the model represented by a fault-aware Kripke structure K, i.e., they represent disjunct sets of capability instances required by the system's tasks to process the resources. For each capability instance $c \in \bigcup_{i=1}^{n} C_i$, the fault $f_c \in F(K)$ represents exactly the loss of capability c without any other effects. In the personalized medicine case study, for example, the capability instances contained in C_1 and C_2 are all instances of the produce and consume capabilities, respectively. All instances of the process capabilities for each ingredient type t_i, on the other hand, are denoted by the sets C_{2+i}; the faults corresponding to these capabilities are thus failing dispenser mechanisms. The maximal fault sets that can possibly be safe are of the form $F(K) \setminus \{ f_{c_1}, \ldots, f_{c_n} \}$ where $(c_1, \ldots, c_n) \in \prod_{i=1}^{n} C_i$. That is, the sets contain all faults except for one of each kind of capability, leaving exactly one capability of each type intact; e.g., one light gray, one medium gray, and one dark gray particulate dispenser continue to work while all others do not.

The heuristic can additionally take advantage of fault subsumption information contained in the analyzed S# model. Instead of only removing a specific fault f_c, all of the faults it subsumes can also be removed at the same time, thereby increasing the cardinality of the fault set. When taking subsumption into account, the maximum safe sets are thus of the form $F(K) \setminus \mathcal{S}(\{f_{c_1}, \ldots, f_{c_n}\})$.

Heuristic 3 (Minimal Redundancy Heuristic)**.** *As early as possible, fault sets of the form $F(K) \setminus \mathcal{S}(\{f_{c_1}, \ldots, f_{c_n}\})$ with $(c_1, \ldots, c_n) \in \prod_{i=1}^{n} C_i$ should be checked to determine whether they are safe for hazard H. If some of the sets turn out to be critical for H, analyze their subsets top-down by decreasing cardinality.*

Similar to the fault subsumption heuristic, the capability types C_i as well as the faults f_c for each $c \in \bigcup_{i=1}^{n} C_i$ in general cannot be determined automatically from a S# model. For self-organizing resource-flow systems based on the metamodel, however, an automatic detection is indeed possible. A manual instantiation of the `MinimalRedundancyHeuristic` for the personalized medicine case study would look as follows in S# for a model instance `model`, taking advantage of C#'s language integrated query functionality in a helper method that selects the ingredient tank defects for an ingredient type:

```
IEnumerable<Fault> GetIngredientTankFaults(Model model, IngredientType type)
   => from dispenser in model.Stations.OfType<ParticulateDispenser>()
      from tank in dispenser.IngredientTanks
      where tank.IngredientType == type
      select tank.TankDefect;

var analysis = new SafetyAnalysis();
analysis.Heuristics.Add(new MinimalRedundancyHeuristic(
   model,
   GetIngredientTankFaults(model, IngredientType.LightGray),
   GetIngredientTankFaults(model, IngredientType.MediumGray),
   GetIngredientTankFaults(model, IngredientType.DarkGray));
```

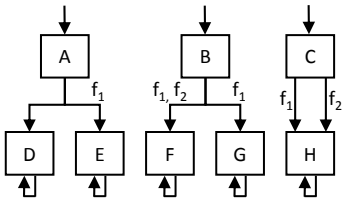
8.2.3 Fault Activation Enforcement

In order to determine whether a fault set $\Gamma \subseteq F(K)$ is safe for fault-aware Kripke structure K and hazard H, the model checker has to enumerate all paths through K. As faults can be activated completely nondeterministically, in the worst case there are $2^{|\Gamma|}$ possible combinations of fault activations to consider in each reachable system state. While the activation minimality of fault-aware Kripke structures reduces analysis effort compared to the common state-based fault modeling approach, larger fault sets still require significantly more effort to check when they are indeed safe for the hazard. But even for critical fault sets, the path proving the set's criticality might take some time to be found. In particular, heuristics are imperfect and might thus suggest large critical fault sets, making analysis slower overall if they guess wrong often enough.

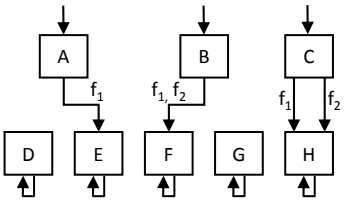
To speed up analyses, DCCAs can be run in a mode where all analyzed faults are forced to be activated: In this mode, S# no longer relies on nondeterministic fault activations but always activates all activatable faults. The idea underlying fault enforcement is that the more faults are activated, the more likely it is that the hazard occurs and the more likely it can be found faster. In a sense, this fault enforcement approach is the dual of fault removal, however, it is unable to uphold DCCA's completeness guarantees as discussed later on. Formally, fault enforcement removes all transitions and initial states of a fault-aware Kripke structure that are not activation-maximal as follows:

Definition 8.3 (Fault Enforcement)**.** Enforcing *all fault activations in a fault-aware Kripke structure* $K = (P, F, S, R, L, I)$ *yields the* fault-enforced *fault-aware Kripke structure* $\mathcal{E}(K) = (P, F, S, R', L, I')$ *with an* activation-maximal *transition relation* $R' = \{(s, \Gamma, s') \in R \mid \forall\, (s, \Gamma', s'') \in R \,.\, s' \neq s'' \rightarrow \Gamma \not\subset \Gamma'\}$ *and* activation-maximal *initial states* $I' = \{(\Gamma, s) \in I \mid \forall\, (\Gamma', s') \in I \,.\, s \neq s' \rightarrow \Gamma \not\subset \Gamma'\}$.

Figure 8.9 shows an exemplary fault-aware Kripke structure that is converted to a fault-enforced one. All transitions leaving some state $s \in S(K)$ with fault activations Γ are removed for which there is another transition leaving s that activates a superset of Γ and analogously for the initial states. In particular, s can only have an outgoing transition without any fault activations if no faults can be activated at all in s. If there are two distinct transitions with different target states activating $f_1 \in F(K)$ and $\{f_1, f_2\} \subseteq F(K)$, only the latter transition is kept. On the other hand, two transitions activating distinct faults $f_1, f_2 \in F(K)$, for example, are both kept as they cannot be unified. In that sense, the transition relation $R(\mathcal{E}(K))$ is activation-maximal as only



(a) A fault-aware Kripke structure with two faults f_1 and f_2 and eight reachable states that represents the actual behavior of the analyzed system.



(b) The fault-aware Kripke structure resulting from the one on the left after fault enforcement with only six reachable states.

Figure 8.9. An illustration of fault activation enforcement: The fault-aware Kripke structure on the left is transformed to the fault-enforced fault-aware Kripke structure on the right by removing all non-activation-maximal transitions. The number of transitions and reachable states decreases, explaining the resulting analysis efficiency improvements as the model checker only has to consider a smaller Kripke structure. However, the behavior of the Kripke structure is changed by fault enforcement as some states such as D and G are no longer reachable.

those transitions activating the largest fault sets remain in the fault-enforced Kripke structure. Nevertheless, fault-enforced Kripke structures are also still activation-minimal, therefore never "activating" a fault that has no effect or duplicating transitions between the same source and target states for which a subset relationship exists.

In contrast to fault injection and fault removal, fault enforcement yields exactly one fault-enforced fault-aware Kripke structure as there is no room for creativity. Compared to fault removal, the operation always succeeds as the resulting Kripke structure is guaranteed to be deadlock-free: For each state $s \in S(\mathcal{E}(K))$, there is always at least one transition $(s, \Gamma, s') \in R(\mathcal{E}(K))$ left as, trivially, $\Gamma \subseteq \Gamma$. For the same reason, at least one initial state remains. As fault enforcement does not introduce any new behavior, the fault-enforced paths are contained in the original Kripke structure's set of paths $paths(K)$, which in turn establishes the usefulness of fault enforcement for more efficient DCCAs of self-organizing systems as explained later on:

Lemma 8.2. *For a fault-aware Kripke structure K and its fault-enforced fault-aware Kripke structure $\mathcal{E}(K)$, $paths(\mathcal{E}(K)) \subseteq paths(K)$.*

Proof. Directly follows from the fact that $I(\mathcal{E}(K)) \subseteq I(K)$ and $R(\mathcal{E}(K)) \subseteq R(K)$. □

The reverse of Lemma 8.2 does not hold for a Kripke structure K with at least one activatable fault: Its fault-enforced counterpart $\mathcal{E}(K)$ does not contain any paths $\varsigma \in paths(K)$ that do not activate any faults, i.e., for all paths $\varsigma' \in paths(\mathcal{E}(K))$, $\mathcal{A}(\varsigma') \neq \emptyset$ in that case. When conducting DCCAs over fault-enforced Kripke structures, the minimal critical fault sets that are found in general differ from the ones found for the original Kripke structure: All critical fault sets discovered with fault enforcement are also critical with nondeterministic fault activations as the path showing the criticality of the set is encountered in both analysis modes; constant activation is one of the many possibilities that exist with nondeterministic choices. The reverse, however, does not hold as a fault set is safe only when there is no path showing its criticality, but with forced fault activation, only a subset of all possible paths is considered. Formally:

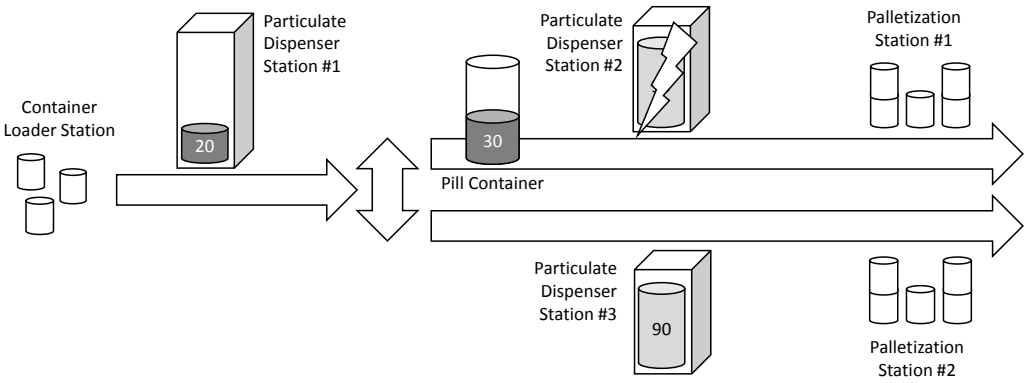


Figure 8.10. A configuration of the personalized medicine case study in which forced and nondeterministic fault activation result in different minimal critical fault sets: The pill container on the upper conveyor belt is almost fully processed, but its processing cannot be completed due to a complete failure of particulate dispenser station #2 that occurs once the pill container reached the upper conveyor belt. From the pill container's location, no other particulate dispenser is reachable and thus the container has to be discarded. However, the system does not have a sufficient amount of dark gray ingredients left to produce a replacement, hence the failure of particulate dispenser station #2 is critical. With forced fault activation, on the other hand, particulate dispenser station #2 would have failed in the beginning, allowing the pill container to be routed to the lower conveyor belt, successfully completing its production process. Consequently, the failure of particulate dispenser station #2 would be safe.

Proposition 8.2. *Let H be a propositional logic formula not referencing any faults $f \in F(K)$. If a fault set $\Gamma \subseteq F(K)$ is critical for hazard H in $\mathcal{E}(K)$, it is also critical for H in K. Conversely, if Γ is safe for hazard H in K, it is also safe for H in $\mathcal{E}(K)$.*

Proof. As safe fault sets are dual to critical ones, it suffices to show the claim for critical fault sets: If Γ is critical for H in $\mathcal{E}(K)$, there is a path $\varsigma \in paths(\mathcal{E}(K))$ such that $\varsigma \models only_\Gamma \, \mathbf{U}^{=} \, H$ with $only_\Gamma$ as in Definition 5.13. Due to Lemma 8.2, $\varsigma \in paths(K)$ and thus Γ is also critical for H in K. □

For many systems, fault enforcement can provide a good approximation of the results with nondeterministic activation. For self-organizing systems, the optimization is particularly interesting as it significantly reduces analysis times as shown in Section 8.2.4. The robot cell case study is even able to profit from the analysis time reductions without sacrificing DCCA completeness when analyzing the tolerable faults: When a fault set Γ is checked for criticality, either the remaining redundancy is sufficient to find a new configuration or the set is indeed critical. If the redundancy suffices, it would also have been enough had the faults been activated nondeterministically as the order of tolerable fault activations is irrelevant for the robot cell case study. For the personalized medicine case study, on the other hand, there are some edge cases where critical fault sets might be missed as explained by Figure 8.10. The key difference between both case studies lies in the ingredient amounts: The personalized medicine case study has capabilities that eventually cannot be used anymore even without fault activations. Consequently, nondeterministic fault activations at inopportune times might result in pill containers for which processing cannot be completed. If the remaining amount of ingredients is

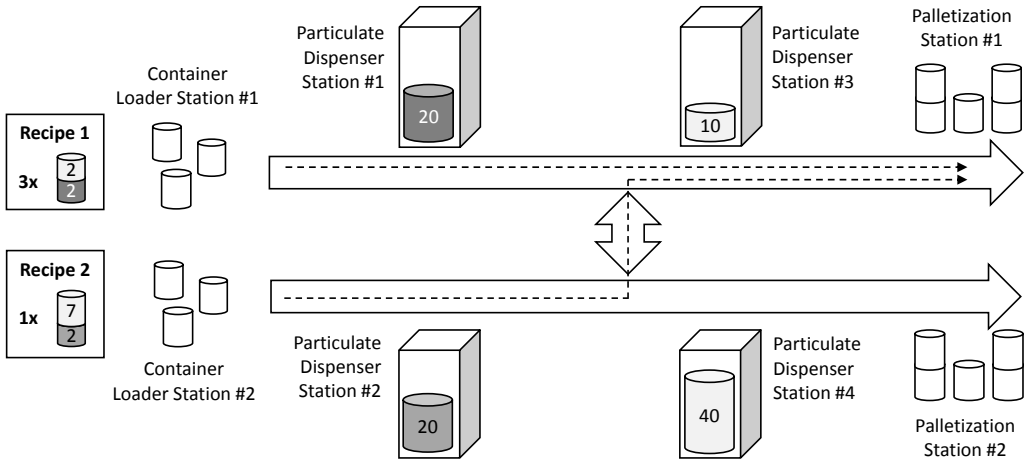


Figure 8.11. A configuration of the personalized medicine case study where ingredient amounts are so low that they are just barely sufficient to process the system's active recipes: Loader station #1 produces pill containers for recipe #1, whereas loader station #2 produces pill containers for recipe #2. The pill containers of each loader follow the dashed arrows, both ending at palletization station #1; for pill containers with recipe #2, such a path might be chosen when the connection between stations #2 and #3 is shorter than the one between #2 and #4, for instance. Due to this routing, recipe #1 cannot be fully processed as station #3 runs out of ingredients. However, had the system been analyzed with station #3 only containing six units of the ingredient, processing would have been completed as the pill containers for recipe #2 would have been routed to station #4.

low, it might also no longer be possible to produce a replacement as the system does not have enough ingredients left. Had the fault activations been enforced from the beginning, the system might have been able to prevent itself from maneuvering into such a situation as illustrated by Figure 8.10.

For design time analyses without hard timing constraints, it is preferable to use the fully nondeterministic fault activation mode in order to benefit from DCCA's completeness guarantees. During the constructive commissioning phase of self-organizing systems, a selection of configurations can thus be thoroughly analyzed. At run time, on the other hand, virtual commissioning requires analyses to be fast, making forced fault activation an acceptable tradeoff. For the personalized medicine case study, the risk of missing a critical fault set of a configuration can further be reduced by analyzing the system with less remaining ingredients than the system actually has available. For example, there might be a rule saying that all ingredient tanks must be refilled as soon as their fill level falls below x percent. The system's safety can then be overapproximated by analyzing the system as if it only had x percent of all ingredients left, making the system seem less safe than it actually is. Due to the refilling rule, the system's ingredients will typically be restocked before the system enters a state for which the overapproximation is no longer valid. As shown by Figure 8.11, however, even this overapproximation is only an estimate that might also overlook some minimal critical fault sets.

All in all, DCCAs with forced fault activations remain a tradeoff between analysis efficiency and analysis completeness. For the robot cell case study, in fact, the optimization

is always valid due to the workings of the system's capabilities and its reconfiguration mechanism. For the personalized medicine case study, on the other hand, there are known cases of missed minimal critical fault sets as exemplified by Figures 8.10 and 8.11. However, none of the configurations evaluated in Section 8.2.4 actually miss a minimal critical fault set with forced fault activations, showing that the gains in analysis efficiency are likely worth the risk of incompleteness. Additionally, DCCAs with nondeterministic fault activation can be conducted while the system is running with the new configuration, using the additional time and the information obtained from the previous, incomplete analyses to efficiently conduct a complete DCCA.

8.2.4 Evaluation of Efficiency Improvements

In order to evaluate the effectiveness of the two heuristics and forced fault activation, several different configurations of the personalized medicine case study are analyzed for the hazard of non-existence of further configurations, considering tolerable faults only. The analyzed configurations differ in the number of stations as well as their connections through conveyor belts and thus in the number of faults that have to be checked. The metrics that are compared are the number of fault sets that have to be checked for criticality as well as the overall time it takes to complete a DCCA for the configurations using either neither of the heuristics, only one of them, or both of them in combination. Table 8.4 shows the results of these analyses; since the heuristics are deterministic, an analysis of standard deviation and other stochastic properties is not necessary.

Fault Set Suggestion Quality. The fault sets suggested by the heuristics generally fall into two categories: Either the heuristics suggest a fault set that is already known to be safe or critical, or they suggest a fault set for which no information is yet available. In the former case, the suggestion is trivially safe or trivially critical, respectively. In the latter case, the suggestion is only helpful when the fault set is actually safe for the hazard, potentially allowing many sets to be skipped that no longer have to be analyzed. The percentages in the "# checked sets" columns of Table 8.4 indicate what fractions of the sets suggested by the heuristics were non-trivially safe, i.e., the fractions of suggestions that actually contributed to the reported analysis time reductions.

For most evaluated configurations, the majority of the fault sets suggested by the heuristics are critical. Averaged over all analyzed configurations, only 7.4% of the fault sets suggested by heuristic 2, 0.2% of those sets suggested by heuristic 3, and 0.1% of the sets suggested by both heuristics combined were non-trivially safe. Phrased differently, the absolute majority of all suggestions do not contribute to the observed analysis time reductions, instead actually increasing analysis times. However, a single non-trivially safe fault set of cardinality n suggested by a heuristic eliminates up to $2^n - 1$ sets from consideration, explaining the effectiveness of the heuristics despite their low suggestion quality. By contrast, the high number of false positives can become problematic when analyzing in nondeterministic fault activation mode as the time required to analyze such a false positive is exponential in the number of faults it contains. In general, the large number of unnecessary checks might negatively impact analysis time, although for the configurations considered in Table 8.4, the heuristics' overall positive effects outweigh the overhead introduced by their mistaken suggestions.

	# stations	# faults	# minimal critical sets	without heuristics # checked sets	without heuristics time	with heuristic 2 # checked sets	with heuristic 2 time	with heuristic 3 # checked sets	with heuristic 3 time	with heuristics 2 and 3 # checked sets	with heuristics 2 and 3 time
A	3	9	9	10	1.4s 1.4s	11 (0 %)	1.2s 1.3s	10 (0 %)	1.2s 1.3s	11 (0 %)	1.2s 1.3s
B	4	14	20	144	2.1s 2.2s	35 (36 %)	1.6s 1.8s	83 (50 %)	1.8s 1.9s	37 (27 %)	1.6s 1.7s
C	6	18	57	1340	5.6s 13.7s	261 (31 %)	3.0s 4.5s	349 (6 %)	3.2s 4.9s	240 (6 %)	3.1s 4.3s
D	6	18	41	3244	10.8s 33.5s	453 (53 %)	3.5s 7.7s	375 (56 %)	3.3s 6.4s	172 (13 %)	2.8s 4.5s
E	6	18	27	3702	10.7s 37.4s	459 (56 %)	3.4s 7.7s	59 (100 %)	2.6s 3.4s	66 (5 %)	2.5s 3.6s
F	7	23	130	76639	2m 52s 65m 42s	1388 (16 %)	5.9s 2m 49s	1813 (7 %)	7.6s 1m 39s	1658 (2 %)	7.2s 1m 30s
G	9	27	81	5.5 mio	3h 5m	35117 (20 %)	1m 35s	324 (100 %)	12.3s	343 (0 %)	15.0s
H	9	27	341	1.3 mio	43m 59s	9518 (19 %)	25.8s	57424 (1 %)	4m 30s	46789 (1 %)	4m 10s
I	10	32	805	56.9 mio	1d 8 h	47246 (5 %)	6m 01s	174976 (0 %)	17m 45s	129214 (0 %)	16m 43s

Table 8.4. Evaluation of nine different configurations of the personalized medicine case study, comparing the number of fault sets that have to be checked for criticality as well as the analysis times. All four combinations of using heuristics 2 and 3 are evaluated, using either the forced or nondeterministic fault activation mode for configurations **A** to **F** and only forced fault activation for configurations **G** to **I**; results for forced fault activations are always shown on the upper lines. For most configurations, one of the heuristics yields significantly better results than the other. The effectiveness depends on the configuration's topology, i.e., the number and circularity of the connections between the stations.

Analysis Efficiency Improvements. For the first six configurations **A** to **F**, the analyses are carried out using both forced and nondeterministic fault activation mode; for the remaining configurations, only forced activations are considered due to the longer analysis times. The differences between both modes become more pronounced as fault counts increase. Both modes yield the exact same minimal critical fault sets in all analyzed configurations, showing that analyses with forced fault activations indeed provide close approximations of complete DCCAs. The general efficiency improvements resulting from the use of the heuristics are substantial in all configurations except for **A**. Even though the heuristics still scale exponentially, they do so at a much lower rate, allowing the largest configuration **I** to be analyzed in roughly 6 minutes in the best case instead of 1.3 days. The time required for the analyses grows faster than the number of analyzed sets as trivially safe or trivially critical sets suggested by a heuristic are not counted as a checked set. Nevertheless, the S# framework still has to briefly consider these sets to discard them. Furthermore, the heuristics also take some time to make their suggestions as they have to consider most of the fault sets that DCCA would otherwise analyze; for the larger models, the total number of suggestions lies in the millions.

The two heuristics have different strengths depending on the topology of the analyzed configurations. For example, configurations **B** and **C** have a circular topology where heuristic 3 is less successful than heuristic 2. Heuristic 3 is better suited when there are more connections between stations, as otherwise its suggestions are mostly critical: Removing all but one instances of some capability is critical when the remaining capability instance is unreachable by the allowed resource flows. For instance, the only difference between configurations **C** and **D** is that in configuration **D**, the circle is connected bidirectionally, immediately resulting in heuristic 3 to yield better results than it did in configuration **C**: 56% of the sets it suggested were non-trivially safe, as opposed to just 6% in model **C**. For highly connected configurations such as **E** and **G**, the latter of which is a complete network, the effect becomes even more obvious, resulting in all suggestions being non-trivial safe sets. In such situations, the combination of both heuristics reduces analysis efficiency. By contrast, configuration **H** again illustrates the weakness of heuristic 3 for sparse networks: Even though the configuration has the same number of stations and faults as configuration **G**, it has significantly fewer connections, consisting of three separate production lines that only have their beginnings and ends connected together. All in all, using both heuristics together provides neither substantial efficiency improvements nor analysis time degradations. System configurations for which no educated guesses based on their topology are possible, combinations of both heuristics are hence a sound optimization strategy.

8.3 Model-Based Testing of Self-Organization Mechanisms

Self-organization is a complex mechanism that requires significant development effort to get right. The restore invariant approach and the metamodel for self-organizing resource-flow systems were developed to provide a standard framework and architecture for amending a system with self-organization concepts in a structured way that avoids common pitfalls [77, 146, 180]. Nevertheless, the actual implementations of the self-organizing systems must still be tested for correctness just like any other

safety-critical system [131, 186]; only tests carried out on the actual system can show adequacy. Development faults such as design flaws or implementation bugs are the typical reasons for functional incorrectness. While for the system's functionality modulo self-organization aspects, that is, the functional system depicted in Figure 8.2 on page 185, traditional testing methods can be used, the observer/controller poses the same challenges during testing as for design time or run time formal safety analyses: It is impossible to exhaustively test all system configurations, as their number is in general unbounded. Additionally, the large number of tolerable faults makes it impracticable to test all of their combinations even for a single configuration. It is therefore necessary to select appropriate test cases systematically and to supply test oracles that are able to determine whether the self-organization mechanisms are correct.

In order to systematically test a self-organization mechanism, the same basic concepts can be used that enable the run time analysis approach introduced in the preceding sections. However, run time analyses by themselves are insufficient to test functional correctness even though continuously monitoring and checking the system for unacceptable minimal critical fault sets can of course also detect development faults. In the presence of development faults, run time DCCAs would likely determine the empty set of faults to be minimal critical, which indeed signals a highly important safety issue that needs to be resolved but does not provide any insights into what the underlying problem actually is. In particular, run time analyses with S# are unable to determine whether indeed no further configuration exists or whether the observer/controller is simply unable to find one due to an implementation bug, for instance. Moreover, tests should be conducted as early as possible in the development process as the cost of fixing architecture, design, or implementation bugs increases the later they are found and fixed during development [161].

The S#-based testing approach for self-organization mechanisms introduced in the following uses model-based back-to-back testing [199]. Back-to-back testing is a standard testing methodology that is, for example, suggested by ISO 26262 [107] for model-based development of safety-critical systems. With back-to-back testing, two implementations of a system are tested against each other in order to determine whether they produce the same outputs for the same inputs. In the following, a S# model as well as the actual implementation of a self-organization mechanism are tested back-to-back, with the S# model's executability allowing it to be viewed as another, potentially more high-level, implementation. The testing approach is model-based as the test cases are derived from the S# model during testing, in particular from the tolerable faults injected into the model. Ideally, the model and the implementation are developed separately by different teams to avoid making the same development faults twice; otherwise, the same faults might be contained in the model as well as in the implementation, in which case back-to-back testing is unable to identify them.

Even though the general testing approach is more broadly applicable, the following discussion only concerns itself with the class of self-organizing resource-flow systems implemented on top of the restore invariant approach. The robot cell case study is used as the running example, also providing the basis for the evaluation of the testing approach in Section 8.3.4. Despite the verification of the result checker's correctness

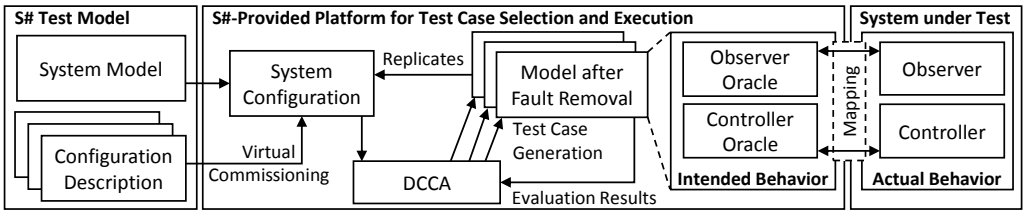


Figure 8.12. Overview of the S#-based testing approach for self-organization mechanisms: The system models and configuration descriptions enable S# to instantiate a specific system configuration, subsequently allowing DCCA to generate the test cases; a test case thus corresponds to a fault set that is checked for criticality. The test cases are automatically executed by S#'s automated DCCA execution algorithm, checking the observer and controller oracles against the behavior of the system under test, i.e., the actual implementations of the observer and controller.

by Nafz et al. [146], it is still necessary to test the actual implementation of the self-organization mechanism: The implementation of the result checker could contain bugs that cannot be identified by the formal verification but only by thorough, systematic testing. Additionally, other self-organizing resource-flow systems might require minor modifications to the underlying metamodel, e.g., to support the ingredient amounts of the personalized medicine case study, potentially invalidating the formal verification as the implementation by design no longer matches the verified model. In this case, testing is imperative to ensure that the modifications do not introduce any bugs.

Figure 8.12 gives an overview of the general testing approach for self-organization mechanisms based on S#. The approach consists of three main parts, each of which is described in further detail in the following sections. The first part consists of the S# test model of the system containing all relevant tolerable faults. For the robot cell case study, for instance, the S# test model corresponds to the model that is used during virtual commissioning. Different system configurations can be instantiated using S#'s flexible model composition and instantiation capabilities. The second part is the test platform provided by the S# framework, using DCCA as the test case generation and execution mechanism; the reduced models that contain only the tolerable faults that DCCA currently checks for criticality represent the test cases that have to be executed. The actual configuration that is tested can be determined at run time when the system is about to operate in the selected configuration. Consequently, virtual commissioning facilitates both run time analyses as well as testing of self-organizing systems, coping with the unbounded number of configurations that exist by only considering those configurations that are actually used. The third part of the approach is the actual system under test, that is, the real-world implementation of the self-organization mechanism that is tested, requiring a mapping between the S# model and the actual system.

8.3.1 Platform for Test Case Selection and Execution

The testing approach repurposes DCCA, using it as its test case generation and test execution environment: For a given system configuration, a complete DCCA is conducted for the hazard of non-existence of further configurations. During the entire time, the S# model validates the configurations computed by the system under test in order to

check their validity; in particular, if no new configuration is computed even though one still exists, an exception is thrown, causing the test case to be classified as failed. Test execution is successful when no exception is thrown during a complete DCCA, i.e., as long as valid configurations exist, the system under test must be able to find them and all computed configurations must satisfy the RIA predicate. The test approach therefore assumes the actual reconfiguration mechanism to always find a solution as long as one exists; genetic algorithms, on the other hand, may fail to do so, which can be an acceptable tradeoff if more thorough techniques such as constraint solving are too inefficient. In that case, the testing approach can be used to judge the quality of the algorithm, identifying situations in which the algorithm is likely to fail.

DCCA can be seen as applying a boundary-interior test selection strategy [99]: Test cases are selected such that the system is at or near the boundary of expected behavioral changes, i.e., a RIA-based self-organizing system changes its behavior drastically whenever the corridor of correct behavior is permanently violated as it cannot continue functioning at all. Thus, DCCA tries to converge towards the boundary of the RIA corridor where the system under test most likely fails to find new configurations. Without the use of heuristics, DCCA's bottom-up search strategy includes many non-boundary test cases, checking thousands if not millions of safe fault sets that are less likely to uncover development faults. With heuristics, on the other hand, DCCA converges more quickly towards the boundary as the heuristics are designed to suggest fault sets that lie as close to the boundary as possible. Figure 8.13 illustrates these test strategies.

8.3.2 System Under Test

The S# test model describes the intended behavior of the system and interfaces with the system under test for evaluation purposes: As the S# framework drives the entire testing process by repurposing DCCA, the S# model must not only be able to reason about the system's intended behavior, but also about the actual behavior of the system under test in order to determine whether the model and the implementation expose the same behavior. For the robot cell case study, the block definition diagram shown by Figure 8.7 on page 192 shows the integration of the actual system under test into the S# model: The MiniZincObserverController class interfaces with the MiniZinc constraint solver [148], representing the self-organization mechanism under test in the form of a MiniZinc constraint problem that is solved using FlatZinc [148]. Whenever the observer triggers a reconfiguration, the S# model invokes MiniZinc with an instantiation of the constraint problem mirroring the state of the S# model, specifying the task to be fulfilled, the number of available agents as well as their respective, still functional capabilities, and the routes the carts can take to transport workpieces between the robots. That is, during test execution, the constraint solver is invoked by the S# model, converting the model's system state into the input format that the constraint solver understands. If MiniZinc finds a solution, the solution is mapped back onto the S# model to determine its validity. If no new configuration is found, the test model checks its oracles to determine whether there indeed is no further configuration, reporting a test failure if the mechanism under test and the test oracle disagree with each other. Consequently, the MiniZinc-based implementation is tested back-to-back with the S# test model.

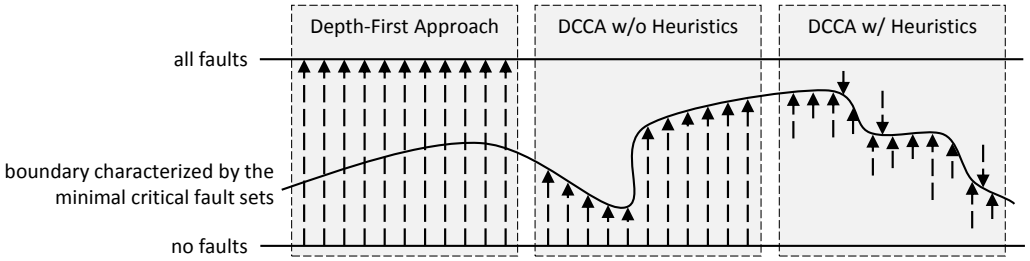


Figure 8.13. Conceptual illustration of three different search strategies for the boundary of the corridor of correct behavior. The system configurations that conceptually lie at the bottom are assumed to have no faults activated whereas at the top, all tolerable faults contained in the S# model of the system are activated. The boundary in between graphically depicts the different minimal critical fault sets that prevent the system from finding a valid configuration. Consequently, valid configurations exist below the boundary while above it, no valid configurations can be found. The depth-first search strategy used by model checking without DCCA would cross the boundary several times, analyzing large numbers of fault sets for which no configuration can possibly exist; such tests only increase test execution time but do not increase test coverage in a meaningful way. DCCA without heuristics, on the other hand, employs a breadth-first search that more quickly converges against the boundary and never crosses it. However, without heuristics, there are still many superfluous tests well within the corridor of correct behavior where development faults of the system under test are unlikely to be discovered. Once heuristics are enabled, on the other hand, DCCA quickly converges against the boundary from both sides, as the heuristics suggest both safe as well as critical fault sets. Since the heuristics not always suggest minimal critical or maximal safe fault sets, some superfluous test cases are still generated, but the overall number of test cases is reduced significantly.

8.3.3 Test Model

The S# test model contains the plants' behavior as well as the test oracles for the observer/controller under test, invoking the latter as explained above. As the system's RIA predicate determines valid system states, the predicate is contained in the S# model in the role of a test oracle in order to determine whether a reconfiguration is necessary, newly computed configurations are valid, and new role allocations are correctly distributed to all agents. For example, the following snippet of the robot cell's S# model shows the I/O consistency and capability consistency constraints modeled in S# as lambda functions for a specific agent instance `agent`; the entirety of all such constraints forms the RIA predicate. The first constraint checks whether the preconditions' ports of the agent's allocated roles match the agent's inputs. The second constraint ensures that the capabilities required by the agent's allocated roles are still available, i.e., the agent has not lost any of its allocated capabilities due to some fault activations.

```
agent.Constraints = new List<Func<bool>>()
{
   () => agent.AllocatedRoles.All(role => role.PreCondition.Port == null ||
       agent.Inputs.Contains(role.PreCondition.Port)),
   () => agent.AllocatedRoles.All(
               role => role.CapabilitiesToApply.All(capability =>
                   agent.AvailableCapabilites.Contains(capability))),
   // Additional constraints omitted...
}
```

Input: robotAgents, cartAgents, tasks
Output: a Boolean value indicating whether a reconfiguration is possible

$m \leftarrow$ GetConnectionMatrix(robotAgents) // transitive closure of all connected robots
for all $t \in$ tasks **do**
  if $\neg\forall c \in t$.Capabilities . $\exists a \in$ robotAgents . $c \in a$.AvailableCapabilities **then**
    return false
  end if
  $A \leftarrow \{a \in$ robotAgents $\mid t$.Capabilities[0] $\in a$.AvailableCapabilities $\}$
  for $i = 0$ **to** |task.Capabilities| $- 1$ **do**
    $A \leftarrow \{a \in m[a'] \mid a' \in A \wedge t$.Capabilities$[i+1] \in a$.AvailableCapabilities $\}$
    if $|A| = 0$ **then**
      return false
    end if
  end for
end for
return true

Algorithm 4. The algorithm that the S# test model uses as its test oracle to determine whether a reconfiguration is possible for a given set of robot and cart agents as well as the tasks to be carried out. The algorithm starts by checking whether all capabilities of the tasks the system has to process are still available; for instance, if at least one task requires a robot to drill a hole, at least one robot must still have the drilling capability available. Subsequently, the algorithm checks whether it is still possible to establish a resource flow between the robots such that the order of tool applications required by the tasks can be adhered to. Therefore, it must be possible for the carts to transport the workpieces between the robots that still have the required capabilities, and they must be able to do so in the order required by the workpieces' tasks.

In fact, the S# model contains two sets of constraints for the test oracle: One is specific for the observer and the other concerns the controller. The observer-related set contains all constraints that describe the corridor of correct behavior, hence their violations must be detected by the observer under test in order to trigger a reconfiguration. Consequently, the constraints for the observer oracle are evaluated once the observer under test actually triggers a reconfiguration, checking whether the reconfiguration is justified; conversely, if the observer under test fails to trigger a reconfiguration even though one is necessary, the system under test is considered to be unable to find a new configuration. Additionally, a superset of the observer constraints must be checked whenever the controller under test computes a new configuration as there are some constraints that affect controller correctness but are otherwise irrelevant for the observer. For example, the allocated roles of the robots must be connected together such that all tasks' required capabilities are applied in the right order. There is no need for the observer to check this constraint, since no environmental influences or fault activations can ever change previously distributed role allocations. Only the tools used by the roles or the connections between agents can fail, for instance, in which case the observer under test must trigger a reconfiguration.

In addition to the controller constraints that allow S# to determine whether the configuration computed by the system under test is valid, it uses Algorithm 4 to check whether the real reconfiguration mechanism finds a new configuration as long as one exists. While this requirement has to be relaxed for self-organization mechanisms based

# Robots	# Carts	∅ Capabilities per Robot	∅ Routes per Cart	∅ Capabilities per Task	# Test Cases	Time
4	3	2.8	2.8	6	131000	9.5h
3	2	1.7	1.7	5	49	0.2m
3	2	3.7	3.7	5	26763	69.3m
3	2	1.7	1.7	5	157	0.4m
3	2	1.7	1.7	8	47	0.8m
5	2	1.6	1.6	5	1577	6.9m
3	4	1.7	1.7	5	369	1.1m

Table 8.5. Overview of the evaluation results for the testing approach based on DCCA without heuristics for seven different configurations of the robot cell case study. The reported times correspond to complete DCCAs, i.e., complete boundary-interior testing of the case study's reconfiguration mechanism. The number of test cases corresponds to the number of fault sets that have to be checked for criticality.

on genetic algorithms, for instance, it is important to check when the implementation is based on constraint solving: This additional check ensures that no development faults such as overconstraint problems [95] are overlooked. That is, there could be an implementation fault that systematically prevents the reconfiguration mechanism under test from finding valid configurations under certain circumstances. In order for S# to determine whether a new configuration still exists, it must itself search for valid configurations using Algorithm 4. Such algorithms might be expensive to execute in both time and space, i.e., they can be a very inefficient test oracle. However, the algorithm is only executed when the system under test fails to find a new configuration, thus it is not executed in all test cases. Additionally, in the situations it is in fact executed in, the system is likely to be almost out of redundancy, in which case there are not many configurations that still exist, reducing the execution time of Algorithm 4.

8.3.4 Evaluation of the Testing Approach

The S#-based testing approach for self-organization mechanisms is evaluated for different configurations of the robot cell case study using an observer/controller based on the MiniZinc constraint solver. Overall testing times reported by Table 8.5 are significantly higher than the run time safety analysis times reported in Section 8.1.5 due to the integration of MiniZinc: The necessary data transformations between the S# test model and the MiniZinc-based system under test as well as the MiniZinc process startup time result in significant overhead of at least 100 milliseconds for each reconfiguration. The algorithm implemented in S#, called `CSharpObserverController` in Figure 8.7 on page 192, on the other hand, typically completes reconfiguration requests around three to four orders of magnitudes faster. Consequently, using MiniZinc for the case study is rather pointless as a special-purpose algorithm exists that solves the problem significantly more efficiently. However, other self-organizing systems might indeed require a constraint solver if no such efficient, special-purpose algorithm exists. The S#-based testing approach can thus also be used for such systems, with the MiniZinc-based evaluation of the robot cell case study demonstrating the applicability of the approach.

Detection of Development Faults. The evaluation of the robot cell case study shows that the testing approach is indeed able to detect various kinds of development faults. As required by the back-to-back testing methodology, the MiniZinc-based implementation of the system was developed separately from the S# model, revealing the following development faults mostly caused by misinterpretations of informal requirements:

- The first development fault results from an imprecise specification of the output provided by MiniZinc: When the newly computed configuration is mapped back to the S# model, the S# model expects the capabilities allocated to an agent to be a zero-based index into the agent's array of available capabilities. MiniZinc, by contrast, returns the zero-based index within the task's sequence of required capability applications.
- The second development fault is caused by a misunderstanding of the meaning of the resource state contained in the pre- and postconditions of the allocated roles. The S# model assumes the state to consist of the capabilities that have yet to be applied on a resource instead of those capabilities that already have been executed. Consequently, for a task requiring the application of $[\mathsf{D}, \mathsf{I}, \mathsf{T}]$, the S# model should have mapped the state to $[\,]$ in the precondition of the first robot and to $[\mathsf{D}]$ in the postcondition if the robot only applies the drill capability. Instead, the precondition's state is set to $[\mathsf{D}, \mathsf{I}, \mathsf{T}]$ and the postcondition's state is set to $[\mathsf{I}, \mathsf{T}]$, resulting in a RIA predicate violation.
- The third detected development fault affects route handling: The MiniZinc implementation interprets routes between robots as direct connections, whereas some of these connections only result from the transitive closure computed by the S# model. In some situations, MiniZinc consequently computes configurations that require a cart to directly connect two robots that would actually require at least one intermediate robot in between; i.e., MiniZinc computes an invalid configuration in this case.
- The fourth development fault also concerns route handling: MiniZinc considers specified routes to be unidirectional even though the S# model means them to be bidirectional. At run time, the fault can thus manifest itself as the system under test being unable to find a new configuration even though at least one still exists; i.e., MiniZinc does not make use of all available redundancy.
- The last uncovered fault results from overly strict restrictions in the MiniZinc model that do not allow the use of intermediate robots in the resource flow. Such robots do not have any allocated roles but simply pass on any incoming workpieces to some other cart. Consequently, MiniZinc fails to find a new configuration in some situations because it is too inflexible with regard to resource routing.

DCCA-based test execution detects the first two development faults with all seven configurations listed in Table 8.5. The other three development faults, by contrast, mainly depend on the topology of the system, requiring a sufficient amount of redundancy to be taken away by tolerable faults in order to maneuver the system into a situation in which the development faults manifest themselves. Thus, the smaller configurations shown in Table 8.5 are unable to uncover the development faults related to route handling.

Efficiency of DCCA-Based Testing. DCCA is repurposed for test case selection and execution in order to decrease the time it takes to detect development faults. The underlying assumption is that such faults are more likely to manifest themselves in

situations in which the system is in the vicinity of the boundary of the corridor of correct behavior, already having used up most of its redundancy reserves. The first two aforementioned faults are so severe that they are discovered almost instantly regardless of which of the search strategies illustrated in Figure 8.13 is used by the testing approach. The remaining three development faults, however, are indeed more likely to occur at or near the boundary. Compared to a depth-first testing approach based on model checking that does not take advantage of DCCA, DCCA-based test execution generally does not reveal the development faults much faster. The biggest exception is the fifth development fault related to the overly strict restrictions of the MiniZinc implementation; in that case, DCCA is able to discover the fault in about 420 seconds while the depth-first approach takes more than two hours. Consequently, more complicated and less likely faults can be uncovered more quickly with DCCA even without the use of heuristics. With heuristics, the time required to find the faults decreases by an additional factor of roughly 2x.

8.4 Towards Analysis and Testing of Adaptive Robot Systems

As exemplified by the robot cell case study, future robot systems become more flexible than their contemporary counterparts by increasing their level of autonomy through self-organization or adaptation. System and software development for robots therefore heads towards component-based architectures [32] to cope with these novel challenges, in particular to support new forms of human-robot collaboration [94]: Detecting human presence and calculating an appropriate reaction is one of the primary challenges for the development of collaborative working environments for robots and humans. To avoid injuries, robot control software has to be able to adapt to hardware failures and new, unforeseeable situations such as external disturbances. New concepts and techniques are required on many different levels to ensure both functional correctness and safety of these systems. For example, revisions of robot safety standards such as ISO 10218 [109] address ongoing developments by updating and renewing regulations as well as safety considerations for human-robot collaboration. To comply with these future standards, scalable, automated, and reliable analysis and testing techniques for autonomous, adaptive robot systems must be developed. The following discusses some first ideas for an integrated analysis and testing approach based on the S# framework that addresses functional quality goals as well as safety aspects.

Component-Based Adaptive Robot Systems. The component-based run time environment for robotics by Vistein et al. [197] is able to adapt the robot control software to new or modified system-level tasks whenever necessary. It supports real-time components that access a wide variety of sensors and actuators such as detection mechanisms for humans or tools that operate on workpieces, respectively. Computational subtasks allow the execution of real-time-sensitive algorithms and continuous behavior with dedicated ports restricting and coordinating the communication between different components. In particular, new computational tasks can be submitted to the run time environment during task execution with synchronization rules stopping running tasks and starting newly submitted ones at the appropriate times. Hence, the run time environment is able to change the composition of active components in order to adapt the overall behavior of the robot system to changing environmental circumstances.

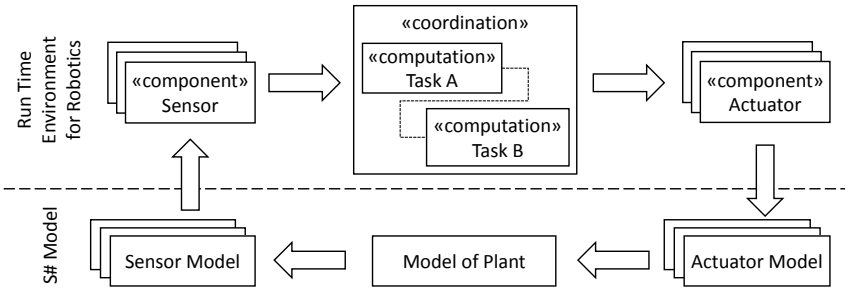


Figure 8.14. An overview of the integration of S# models and the run time environment for robotics. The usual feedback cycle between plants and controllers enables the system's overall adaptivity. The upper part of the figure shows the controller, i.e., the actual component-based robot control software whereas the lower part depicts the plant, sensors, and actuators modeled in S#. The sensor and actuator components used by the controller support both the integration of real hardware as well as the connection to the S# model. Faults must be integrated into the S# model in order to allow S# to assess system safety.

Integrating Component-Based Robotics and S#. The analysis and testing approach for adaptive robot systems integrates the component-based robot run time environment into the model checking-based verification and safety analysis techniques provided by S# as illustrated by Figure 8.14: The integration enables virtual commissioning, similar to how the MiniZinc constraint solver can be integrated into S#-based analyses. The S# model describes behavioral and structural aspects of the controlled plant as well as the sensors and actuators available to the controller. Additionally, S# handles the activations of relevant faults in order to analyze and test the actual control software's reactions to these disturbances. Thus, the S# parts of the model test-drive the actual controller software using S#'s model checker, enabling systematic, explorative, model-based testing of adaptive, component-based robot systems.

The two parts of the integrated model must bridge a gap between the different run time environments and programming languages; like most embedded systems [119], the robotic control software is implemented in C++ and runs in a real-time critical context. The S# parts of the model can make use of standard technologies such as remote method invocation in order to interface with the device drivers of the actual robot controller, for example. However, abstractions of the plant behavior are required to cope with the infinite amount of possible human-robot interactions. It therefore seems unlikely that fully exhaustive model checking is possible for these models, in particular due to the low cycle times of the robot controllers in the range of a few milliseconds. Bounded model checking might have to suffice, at least facilitating exhaustive analyses for short periods of time that thoroughly check a few seconds of actual robot behavior. These restrictions reduce S#'s model checking-based analyses to a form of non-exhaustive testing, which, however, still enables analyses of both functional correctness as well as safety concerns through incomplete, yet sound, DCCAs.

Testing Functional Correctness & Safety of Robot Systems. The integrated analysis and testing approach enables tests that verify and validate the robot control software's functional correctness and can make use of DCCA for safety assessment. However, these DCCAs are incomplete, only uncovering potential safety issues for small fractions

of thoroughly analyzed behavior. For the selection of appropriate test cases, two test selection strategies seem promising: Human workflows in the collaborative working environments are typically standardized and repetitive, making it possible to derive usage profiles with low variance that contain the most likely interactions. As hazards often result after unpredictable and unforeseeable situations, random testing can be used to reach a certain base level of test coverage for the remaining, less likely, potentially unintentional and unsafe human-robot interactions. Overall, the combination of these two test selection strategies provides high test coverage for highly significant scenarios, at the same time noticeably reducing the number of test cases without completely disregarding more unlikely safety-critical situations. Full test coverage, while desirable, cannot be achieved due to the infinite amount of possible human-robot interactions.

Furthermore, it is necessary to introduce a test oracle into the S# model similar to the oracles for the observer/controller when testing self-organizing mechanisms. The test oracles classify the test results as either correct or incorrect and safe or unsafe, respectively. Due to the high amount of different system states and behaviors, it is infeasible to enumerate all of them, making it necessary to use predicates to specify valid states and behaviors at a more abstract level similar to the RIA predicate. Subsequently, these predicates can be evaluated during test execution to determine whether the tests fail or succeed [50]. Moreover, mathematical models are required to describe the nominal behavior of the robot system using, for instance, differential equations to describe the robots' dynamics. Especially for the latter, the S# modeling language must be extended to support continuous behavior; alternatively, adequate abstractions must be found to map the robot dynamics to S#'s discrete-state, discrete-time model of computation.

8.5 Related Work

Modeling Language Expressiveness. Safety analysis tools such as Scade, VECS, the Compass toolset, HiP-HOPS, and many others [2, 135, 151, 181] could in principle also be used to model and analyze self-organizing systems. Compared to the S#-based approach, however, there are several distinct disadvantages related to the level of expressiveness of the respective modeling languages: The languages do not feature the same flexibility with regard to model instantiation and composition that can be used for the S# models. Consequently, their lack of a built-in run time model adaptation and instantiation mechanism would have to be compensated with additional tools to mirror the analyzed system's actual state onto the models used for analysis. Moreover, the expressiveness provided by C# is important for the creation of adequate and comprehensible models. In particular, the metamodel for self-organizing resource-flow systems can be directly mapped to S# using its object-oriented features as well as its state machine-based behavioral specification mechanism. Furthermore, the reconfiguration algorithm can be conveniently expressed using regular control flow-based constructs and external tools such as the MiniZinc constraint solver can be integrated into the S# model both for formal analyses as well as testing. The ability to simulate and debug S# models using C#'s and Visual Studio's sophisticated debugging engine cannot be underestimated when modeling such complex case studies; without good debugging support, finding and understanding modeling faults is significantly harder.

Alternative Analysis Techniques. The underlying analysis techniques of the aforementioned safety analysis tools do not allow heuristics to be specified that would enable them to efficiently rule out large amounts of safe fault sets. One of the reasons for their inability to do so is their tight integration into specific model checkers [26, 30, 122], whereas the algorithm that drives S#'s DCCA implementation is completely separated from those parts of S# that do the actual model checking. Thus, S#'s DCCA implementation and the heuristics could be used for analyzing NuSMV models, for instance, provided that a NuSMV backend is implemented that interfaces between S#'s DCCA integration and a NuSMV model of a self-organizing system. Additionally, the safety analysis techniques by Bozzano et al. [26, 30] that FSAP/NuSMV, xSap, and the Compass toolset are built upon, cannot take advantage of heuristics without giving up analysis soundness and completeness: Due to their underlying formalization, the techniques can only assume the monotonicity of critical fault sets, i.e., they have to make certain assumptions about the structure of the models as well as the modeled faults and their effects in order to take advantage of monotonicity. DCCA, by contrast, is defined in a way such that monotonicity generally holds regardless of the model that is analyzed, thus making it possible to use the heuristics without affecting soundness or completeness.

Adaptive DCCA. A variant of DCCA was developed for the analysis of self-organizing systems called Adaptive DCCA [67]. It is claimed that the original DCCA cannot be applied to self-organizing systems as hazards may sometimes only occur temporarily, namely when the RIA predicate is temporarily violated before a subsequent reconfiguration results in a valid configuration again. Consequently, Adaptive DCCA extends the original CTL-based notion of fault set criticality to self-organizing systems by allowing the hazard to occur multiple times before it is actually considered to be occurring. Formally, Adaptive DCCA checks the CTL formula $K \models only_\Gamma \, \mathbf{EU}^{=}(\mathbf{EG}(H \wedge only_\Gamma))$ in order to determine whether fault set Γ is critical for a hazard H with $only_\Gamma$ defined as usual. Instead of a single occurrence of H, H must at some point occur continuously without any additional activations of faults $f \in F(K) \setminus \Gamma$. Adaptive DCCA therefore requires all faults $F(K) \setminus \Gamma$ to be activation-independent in K as otherwise the existence of a path in which H occurs and subsequently $only_\Gamma$ holds forever cannot be guaranteed. If activation independence is not given, Adaptive DCCA might overlook some minimal critical fault sets. By contrast, the S#-based analysis approach for self-organizing systems uses the regular definition of safe fault sets by analyzing for different hazards than Adaptive DCCA: At the system-level with intolerable faults, hazards are damaged or incompletely processed workpieces. For tolerable faults, the hazard is not a temporary violation of the RIA predicate but rather the reconfiguration mechanism's inability to find further configurations. Consequently, S#-based analyses only require activation independence to use the fault removal optimization but it has otherwise no effect on analysis soundness or completeness.

VECS-Based Safety Analysis of the Robot Cell Case Study. Using Adaptive DCCA, Güdemann [67] performed an evaluation of a highly abstract model of the robot cell case study modeled with VECS [135], only analyzing a single system configuration with three robots, two carts, and each robot having the drill, insert, and tighten capabilities; each robot must have exactly one role with only one capability allocated. Nine faults were

considered that correspond to the failures of each of the nine capabilities contained in the system. The model is specific for the single configuration and cannot be instantiated with different numbers of robots, carts, workpieces, or tasks that require other capabilities to be applied. Additionally, the reconfiguration mechanism is modeled by allowing the model checker to choose any configuration, requiring an LTL-encoded constraint so that the model checker ignores invalid configurations. As Adaptive DCCA is based on CTL and VECS' model checking backend NuSMV does not support LTL constraints when checking a CTL formula, DCCAs must be carried out manually using the Cadence SMV model checker that supports LTL constraints during CTL model checking. The analyzed configuration has 15 minimal critical fault sets that were found in slightly over four minutes. Analyzing the same configuration with S# only takes a couple of seconds without heuristics; however, fewer minimal critical fault sets are found as the S# model allows zero, one, or more roles for each robot, increasing the system's overall fault tolerance and therefore even requiring more fault sets to be checked for criticality, as there are more safe fault sets compared to the VECS model.

Models@run.time. Virtual commissioning for run time analyses based on S# resembles the models@run.time approach. It also synchronizes the models and the actual adaptive systems at run time in order to analyze their functional correctness and various quality attributes [7], including the use of probabilistic model checking for the run time evaluation of system resilience [45]. However, a systematic approach for run time safety analyses has not yet been developed [19]. The run time models allow an adaptive system to reason about what-if scenarios without affecting the actual software or hardware components, similar to conducting DCCAs for potential future configurations while the system is still within the corridor of correct behavior. The models@run.time approach proposes a new modeling language and interpreter for model execution, whereas S# builds upon the established .NET framework and C# programming language, using .NET's type system as its meta-metamodel. Thus, S#'s meta-metamodel is specifically designed for efficient program execution and run time object composition, both of which form the basis of S#'s support for run time model composition and instantiation. Other research directions in the area of models@run.time concern run time safety certification [192], which, however, lies outside the scope of this thesis.

Optimization Strategies for Run Time Analyses. The strategies proposed by Gerasimou et al. [69] for increasing the efficiency of run time analyses are particularly relevant for S#'s run time analysis approach: The suggested caching strategy is similar to S#'s maximal safe fault sets heuristic which makes recomputing minimal critical fault sets more efficient by reusing previously computed ones as likely candidates for the new ones. The proposed lookahead strategy precomputes analysis results for the most likely successor configurations before the next reconfiguration has even started. The same idea is supported by the S#-based approach, allowing the virtual commissioning phase to be skipped when the minimal critical fault sets of the next configuration are already precomputed. Additionally, Gerasimou et al. [69] relax optimality requirements for reconfigurations in order to noticeably increased analysis efficiency. Forced fault activation is a related idea. In a similar vein, it would also be possible to use statistical model checking instead of fully exhaustive model checking. With statistical model checking,

only the most problematic minimal critical fault sets would be computed instead of conducting complete DCCAs, which might be an acceptable tradeoff to reduce analysis times. In contrast to forced fault activation, statistical model checking can provide statistical evidence for the satisfaction or violation of the analyzed property [126]. Consequently, statistical model checking can yield the wrong result just like DCCAs with forced fault activation, but it is able to limit the probability of making an error.

Run Time Probabilistic Model Checking. Filieri et al. [63] suggest to split up analyses of adaptive systems into two steps in order to facilitate efficient probabilistic model checking at run time: At design time, symbolic expressions are generated from the development models of the analyzed system that are parameterized so that they can be efficiently evaluated at run time when the concrete parameter values are known. These expressions can also be used for sensitivity analyses, for instance, allowing the system to reason about the impact that changes to the parameters have on the overall system. However, large changes to the structure of a self-organizing system might require run time recomputations of the parameterized expressions, decreasing the effectiveness of the overall analysis approach.

Summary and Outlook. Self-organizing systems often have an unlimited number of possible system configurations, making exhaustive safety analyses impossible at design time. The virtual commissioning approach presented in this chapter takes advantage of S#'s model composition capabilities to move safety analyses to run time, only analyzing those configurations that a self-organizing system actually executes at some point. Moreover, faults are classified as either tolerable or intolerable subject to the ability of the self-organization mechanism to work around their activations. This separation allows multiple DCCAs with smaller fault sets to be carried out instead of a single DCCA considering all faults; as DCCA has exponential complexity in the number of faults, analysis times can improve significantly. Additionally, the two safe fault sets heuristics for self-organizing resource-flow systems presented in this chapter further reduce DCCA times noticeably. For the hazard of non-existence of further configurations and system designs based on the RIA approach, the minimal critical fault sets characterize the corridor of correct behavior, showing potential weak spots of a system where it is likely to run out of redundancy. Moreover, DCCA can be used as a test case selection strategy and execution platform for model-based testing of the functional correctness of self-organization mechanisms and adaptive robot systems.

Despite the progress made in this thesis towards a run time safety analysis approach for self-organizing systems, there still are many open research questions and further efficiency optimization opportunities. For example, the two heuristics presented in this chapter are highly effective for the personalized medicine case study, but less effective for the robot cell case study due to the more flexible routing possibilities of the carts. Consequently, additional heuristics must be developed. Furthermore, the maximal safe fault sets heuristic from Chapter 6 must be improved to cope with the large number of minimal critical fault sets that self-organizing systems typically have. Additionally, the overall integration of the virtual commissioning approach with real systems has not yet been attempted at the time of writing; however, the individual steps of the analysis approach such as run time safety analyses and model-based testing of self-organization mechanisms are fully developed in this thesis. The use of the S# framework and its analysis capabilities for adaptive robot systems has only been roughly sketched out so far with many open research questions remaining, e.g., the effectiveness of usage profiles as a test selection strategy is unknown and a way to adequately abstract from complex human behavior and human-robot interactions must be found.

Summary. This chapter summarizes the three main contributions of this thesis in order to give an overview of the achieved outcomes: A systematic modeling approach for safety-critical systems based on executable models is introduced, new formalisms and techniques for efficient formal safety analyses using explicit-state model checking are defined, and an analysis and testing approach for self-organizing systems carried out at run time is suggested based on the unified model analysis features developed for the S# framework. Furthermore, the evaluation results for the case studies are discussed and an outlook to future work is provided.

9 Conclusion and Outlook

The title of this thesis, Design Time and Run Time Formal Safety Analysis using Executable Models, already suggests the three main areas that all of the contributions presented in the preceding chapters fall into: The formal foundations summarized in Section 9.1 such as fault-aware modeling and specification, fault injection, as well as fault removal DCCA enable efficient explicit-state formal safety analyses. The analyses are carried out on executable models with the help of the S# modeling and analysis framework for safety-critical systems as recapitulated by Section 9.2. Adequate S# models of a system are created by following a systematic modeling approach that emphasizes fundamental fault modeling concepts, bringing forward standard software development best practices to safety assessment. S#'s model execution approach not only unifies model checking and model simulations, it also supports the run time applicability of the formal analysis techniques in addition to design time analyses; Section 9.3 summaries the evaluation results of the formal analysis techniques and the S# framework for the case studies. Lastly, an outlook to future work is given in Section 9.4.

9.1 Efficient Formal Safety Analysis

Fault-aware modeling and specification as introduced in Chapter 5 forms the basis for efficient formal safety analyses using DCCA. The core concept of fault activations is directly encoded into fault-aware Kripke structures, allowing fault injection to be defined such that conservative extension is guaranteed, i.e., the nominal behavior contained in a model is not changed by the addition of off-nominal behavior. Fault removal allows faults to be removed without affecting the remaining behavior of the model, in particular allowing previously injected faults to be removed in such a way that a Kripke structure

results that is behaviorally equivalent to the original one. These main achievements of fault-aware modeling and specification can be taken advantage of to improve DCCA efficiency by removing those faults from a model that are irrelevant during a check for criticality. Furthermore, the conceptual improvements made to DCCA make it compatible with a broader variety of analysis tools and increase its practical usability.

Applicability of the Techniques. The formal foundations capture the formal requirements for fault injection and fault removal in a tool-independent way, indirectly describing the allowed changes that most safety analysis tools make during fault injection. Consequently, different analysis tools such as S#, VECS, the Compass toolset, or FSAP/NuSMV can all define their own notions of fault injection and model extension, but in the end, they all have to satisfy the requirements presented in this thesis. On the other hand, if these tools can guarantee activation independence of the analyzed faults, they can also take advantage of the fault removal variant of DCCA, provided that they can remove faults from a model without too much overhead. In particular, fault removal introduces absolutely no overhead in the S# framework.

Tool Independence. The formal contributions made by this thesis are independent of the high-level modeling languages such as S# or others that are used for modeling: As long as these languages support a transformation to fault-aware Kripke structures, they inherit all of the formal foundations, efficiency improvements, and analyzability with DCCA. Such transformations are not necessarily required to be carried out by the actual implementations of the tools as shown for the S# framework: Model execution in conjunction with on-the-fly model checking algorithms avoids the explicit creation of any Kripke structures during analyses, thereby further improving analysis efficiency.

Model Checking Efficiency. For safety-critical systems, faults add a huge number of additional states when using the common state-based fault modeling approach, impeding the use of model checking-based analysis techniques for larger systems. Fault-aware modeling and specification helps to significantly decrease the impact of fault injection on the number of reachable states and transitions by focusing on fault activations instead of fault persistency. Notably, transient faults are the general case subsuming all other forms of persistency; with the S# framework, they result in optimal fault-aware Kripke structures in the sense that no overhead is introduced, i.e., all additional states and transitions are absolutely necessary to describe their effects.

DCCA Efficiency. Even if general model checking-based analyses are very time-consuming for large models, it might still be possible to carry out DCCAs reasonably quick for two reasons: Firstly, the fault removal optimization reduces the number of faults that have to be considered when small fault sets are checked for criticality, thus it is likely that small fault sets can indeed be checked for and found quickly. Secondly, DCCAs are carried out bottom-up using an anytime algorithm that can be aborted at any time, yielding a correct albeit incomplete set of minimal critical fault sets. Even incomplete DCCAs can still be useful for safety assessment: The minimal critical fault sets that are indeed identified by them are often the most relevant ones, as smaller fault sets typically have a larger impact on the overall occurrence probability of a hazard. Thus, they highlight the most severe safety issues of a system.

9.2 Systematic Modeling and Unified Model Execution

The S# framework supports the creation of executable models using the C# programming language and the .NET run time environment. Systematic modeling guidelines with a particular emphasis on fault modeling concepts foster the creation of adequate S# models of both nominal and off-nominal behavior. The underlying model of computation takes advantage of the assumption of zero execution time to reduce both modeling complexity and analysis times. The main difference between the S# modeling and analysis framework and other formal safety analysis tools is the executability of its models and the unified model execution approach that underlies all analyses. Especially for complex case studies such as the self-organizing systems considered in Chapter 8, the S# framework has a competitive edge over other tools for safety analysis: Development, debugging, testing, and simulations of S# models are tightly integrated with formal analysis techniques. In particular, S# makes it possible to use various .NET tools, including Visual Studio, throughout the entire model development and analysis process.

Expressive & Modular Modeling Language. S# provides a component- and object-oriented domain specific language that supports the systematic modeling approach introduced in Chapter 3. It includes comprehensive support for fault modeling as well as flexible model composition capabilities for design variant exploration or virtual commissioning, for instance. Modeling with S# does not feel different from programming with C#, as most C# language features and .NET libraries are supported by S# as well. In particular, the S# modeling language supports the specification of both control flow- and state machine-based behavior as well as mixtures thereof; the former is especially useful to model reconfiguration algorithms of self-organizing systems.

Efficient Model Checking & Formal Safety Analyses. The integration of the formal analysis techniques into the S# framework reduce model checking times by up to three orders of magnitude, in general making explicit-state model checking of high-level S# models competitive with symbolic analysis techniques operating on low-level models. The S# framework conducts DCCAs automatically and efficiently through zero-overhead fault removal. Systems with many or large minimal critical fault sets can additionally specify heuristics that help DCCA to find the minimal critical fault sets considerably faster. Consequently, formal safety analyses of S# models are highly efficient despite the high-level modeling concepts they can use to comprehensively model complex case studies such as the self-organizing robot cell.

Simulations, Tests, Debugging, and Visualizations. The S# framework supports model development through model simulations, model tests, and model visualizations. Moreover, the S# framework seamlessly integrates into standard .NET tools: The C# debugger can be used to diagnose model issues during model simulations as if regular C# programs were debugged. Furthermore, Visual Studio's refactoring capabilities can be used during model creation, whereas UI designers are helpful to create model visualizations. Collaborative modeling in a team is also supported by standard source control software such as Git and build servers that enable continuous integration, automatically validating changes with regression tests for the models. The availability of such tools becomes more important with increasing case study complexity.

Implementation Effort. S#'s unified model execution approach allows large parts of the model execution code to be shared between model simulations and model checking in order to guarantee semantic consistency. Consequently, the framework itself is implemented in relatively few lines of code considering its feature set: At the time of writing of this thesis, the core library that facilitates model execution and all kinds of analyses has about 8500 lines of C# code while the extension library for the C# compiler consists of roughly 3900 lines of C# code. Both libraries are tested with about 640 tests, most of which with multiple test assertions; some of these tests are integration tests that dynamically compile and execute entire S# models. All in all, there are about 22500 lines of test code for S#'s compiler and its core library.

9.3 Design Time and Run Time Safety Analysis

The six case studies introduced in Chapter 2 are modeled and analyzed with the S# framework to demonstrate the applicability of both the formal techniques as well as the S# framework for analyses at design time and at run time. With S#, it is not only possible to model and analyze classical reactive systems such as the pressure tank or the height control, but complex self-organizing ones like the robot cell as well. To cope with the inherent complexity of the case studies, the models take advantage of S#'s systematic, component-oriented modeling approach while analyses benefit from fault-aware modeling and specification, fault removal DCCA, and safe fault sets heuristics.

Modeling Language Expressiveness. The systematic modeling approach presented in Chapter 3 and the expressiveness of the S# modeling language help to cope with the complexities of the case studies, overcoming the modeling-related challenges mentioned in Chapter 2: For the height control case study, multiple design variants can be modeled separately and composed together as demonstrated in Chapter 7. Furthermore, S# facilitates the convenient instantiation of different configurations of the self-organization case studies. In particular, the reconfiguration algorithms of these two case studies can be naturally modeled as algorithms instead of necessitating, for instance, a convoluted state machine-based specification. The railroad crossing case study's two physically separate controllers profit from the systematic modeling approach, whereas the hemodialysis machine case study uses S#'s modeling expressiveness to describe the blood and dialyzing fluid flows between the individual components of the machine.

Tool Support. Complex case studies necessitate better tool support, firstly to help with the creation of the models and secondly to assist with their analyses: Code completion and refactoring tools are invaluable for initial modeling and subsequent changes and improvements that typically have to be made at some point. Regression testing helps to ensure that changes to a model do not have unintended side effects, while model debugging helps with uncovering the causes for such side effects when they inevitably occur. The S# framework inherits all of this tooling from its integration into Visual Studio and the standard .NET tool chain.

Analysis Efficiency. Chapter 2 identifies analysis efficiency as one of the main challenges for all case studies except for the very simple pressure tank. The S# framework and the underlying formal analysis techniques reduce analysis times in several ways:

Fault-aware modeling and specification reduces the number of situations in which faults have to be considered during model checking. For self-organizing systems, virtual commissioning is used to cope with the unbounded number of possible system configurations in addition to a modularization of safety analyses by distinguishing between tolerable and intolerable faults. The fault removal variant of DCCA considerably reduces safety analysis times, typically finding all minimal critical fault sets very quickly; the majority of the time is spent in verifying that the remaining fault sets are indeed safe. The safe fault sets heuristics are specifically designed to improve this last part of a DCCA by reducing the number of fault sets that have to check for criticality.

9.4 Outlook

Even though this thesis provides a comprehensive approach for systematic design time and run time formal safety analyses of executable models, many open research questions about a variety of additional aspects remain that also are of relevance in the context of safety assessment. The following provides an outlook to some future work that could be tackled based on the contributions of this thesis.

Extending the Model of Computation. The model of computation underlying S# models allows for efficient analyses without being overly restrictive as demonstrated by the evaluations of the case studies. However, the S# framework cannot directly express the notion of time, requiring the explicit addition of timers when time-sensitive behavior must be modeled. But such timers are inefficiently treated as regular components; dedicated, more efficient analysis techniques based on clock regions [44] exist that can potentially be integrated into S#'s model execution approach. Conceptually, explicit support for time is similar to fault-aware modeling and specification in the sense that an important concept is explicitly represented semantically in order to reduce analysis times. Similarly, it might be beneficial to make discretizations of some forms of continuous behavior automatic to increase model comprehensibility and analysis efficiency.

Support for Probabilistic Safety Analyses. Minimal critical fault sets make it possible to devise additional safety measures to increase the analyzed system's overall safety. Consequently, the completeness guarantee by DCCA is of high importance during system development. However, the actual probability for the occurrence of a hazard is also of great interest, both during development as well as for safety certification [80, 186]. Therefore, probabilistic analysis techniques are currently being integrated into the S# framework: Similar to S#'s integration of the LTSmin model checker for non-probabilistic analyses, probabilistic analyses integrate the explicit-state MRMC model checker [117]. Probabilities can be specified for fault activations that are tracked during model execution. In this analysis mode, a fault-aware Markov chain is conceptually induced that inherits the state and transition count reductions from fault-aware Kripke structures for transient faults; for other kinds of persistency, state-based fault modeling must be used as otherwise hazard probabilities would be changed inappropriately.

Safety Analysis at Run Time for Self-Organizing Systems. Safe fault sets heuristics are an important step forward to cope with the inherently high level of redundancy of self-organizing systems. The two heuristics presented in Chapter 8 work well for

the personalized medicine case study, but the more complex interconnections of the robots through the flexible routing possibilities of the carts require some additional heuristics for the robot cell case study. Furthermore, it might be beneficial to more tightly integrate the heuristics with the analyzed models: At the moment, the heuristics can only reason about faults when S#'s DCCA algorithm informs them about some analyzed fault set to be either safe or critical. If the heuristics also had access to the analyzed system configurations, they might be able to improve their suggestions; for example, a configuration that only uses some of the agents, capabilities, and routes immediately implies that all other agents, capabilities, and routes are redundant, hence the corresponding fault set is safe. Furthermore, it is an open research question whether the knowledge of the minimal critical fault sets can be used to guide the system's reconfiguration mechanism such that the mean time to failure increases. That is, the minimal critical fault sets could be part of the reconfiguration mechanism's inputs for the computation of the next system configuration, allowing it to avoid configurations that use up the remaining redundancy sooner than other configurations that could have been chosen alternatively.

Safety Analysis of Robot Systems. Adaptive robot systems pose a significant challenge for safety analysis approaches: Robots become more autonomous, yet they are expected to cause no harm for people working near them or interacting with them directly. From a technical standpoint, the integration of a robot controller into a S# model does not seem to be particularly challenging, working in a similar way to the integration of the MiniZinc constraint solver for the S#-based testing approach of reconfiguration mechanisms. However, there are several open research questions concerning the overall analysis and testing approach: Firstly, adequate abstractions must be found for the controlled plants as well as the behaviors of the interacting humans. Secondly, high-impact test cases must be selected given that it seems unlikely to expect complete analyses to be possible for such systems; hence, the S# framework degenerates into an automatic testing tool as for the model-based testing approach for self-organization mechanisms. Thirdly, it is not obvious how the hazards to be analyzed are supposed to be specified in the first place. Moreover, all of these challenges have to be tackled in the context of the approaches and techniques used for analyses of self-organizing systems due to the increasing adaptivity required for future robot systems.

Summary. Despite all of the aforementioned open research questions, this thesis provides several notable contributions in the context of model-based safety analysis at design time and at run time: It advances the formal foundations, increases safety analysis efficiency, and integrates a systematic modeling approach for safety-critical systems based on executable models with these formal analyses through unified model execution. Compared to other safety analysis tools and techniques, the S# framework and all of the supporting contributions provide a unique and comprehensive approach for model-based safety analysis, combining the strengths of rigorous formal analyses with an expressive modeling language supported by state-of-the-art tooling.

Bibliography

[1] N. Aan de Brugh, V. Nguyen, and T. Ruys. MoonWalker: Verification of .NET Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 170–173. Springer, 2009.

[2] P. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing Safe, Reliable Systems Using Scade. In *Leveraging Applications of Formal Methods*, pages 115–129. Springer, 2006.

[3] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[4] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A Model Checker for Concurrent Software. In *Computer Aided Verification*, pages 484–487. Springer, 2004.

[5] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *IEEE Transactions on Computers*, 42(8):913–923, 1993.

[6] A. Armoush. *Design Patterns for Safety-Critical Embedded Systems* . PhD thesis, RWTH Aachen University, 2010.

[7] U. Aßmann, S. Götz, J.-M. Jézéquel, B. Morin, and M. Trapp. A Reference Architecture and Roadmap for Models@run.time Systems. In *Models@run.time: Foundations, Applications, and Roadmaps*, pages 1–18. Springer, 2014.

[8] J. Atwood. Falling Into The Pit of Success. `https://blog.codinghorror.com/falling-into-the-pit-of-success/`, 2007. Accessed: 2016-06-10.

[9] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *Dependable and Secure Computing*, 1(1):11–33, 2004.

[10] M. Azimi, C.-T. Chou, A. Kumar, V. Lee, P. Mannava, and S. Park. Experience with Applying Formal Methods to Protocol Specification and System Architecture. *Formal Methods in System Design*, 22(2):109–116, 2003.

[11] S. Bacherini, A. Fantechi, M. Tempestini, and N. Zingoni. A Story About Formal Methods Adoption by a Railway Signaling Manufacturer. In *Formal Methods*, volume 4085 of *LNCS*, pages 179–189. Springer, 2006.

[12] M. Bacic. On hardware-in-the-loop simulation. In *Decision and Control*, pages 3194–3198, 2005.

[13] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[14] M. Banzi. *Getting Started with Arduino*. O'Reilly Media, 2009.

[15] M. Batteux, T. Prosvirnova, A. Rauzy, and L. Kloul. The AltaRica 3.0 Project for Model-Based Safety Assessment. In *Industrial Informatics*, pages 741–746. IEEE, 2013.

[16] M. Becker, S. Kemmann, and K. Shashidhar. Integrating Software Safety and Product Line Engineering using Formal Methods: Challenges and Opportunities. In *Software Product Line Conference*, volume 2, pages 129–136, 2010.

[17] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *Quantitative Evaluation of Systems*, pages 125–126. IEEE, 2006.

[18] M. Benedetti and A. Cimatti. Bounded Model Checking for Past LTL. In *Tools and Algorithms for the Construction and Analysis of System*, pages 18–33. Springer, 2003.

[19] A. Bennaceur, R. France, G. Tamburrelli, T. Vogel, P. Mosterman, W. Cazzola, F. Costa, A. Pierantonio, M. Tichy, M. Akşit, P. Emmanuelson, H. Gang, N. Georgantas, and D. Redlich. Mechanisms for Leveraging Models at Runtime in Self-adaptive Software. In *Models@run.time*, pages 19–46. Springer, 2014.

[20] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[21] F. Bitsch. Safety Patterns — The Key to Formal Specification of Safety Requirements. In *Computer Safety, Reliability and Security*, volume 2187 of *LNCS*, pages 176–189. Springer, 2001.

[22] E. Börger, N. Fruja, V. Gervasi, and R. Stärk. A High-level Modular Definition of the Semantics of C#. *Theoretical Computer Science*, 336(2-3):235–284, 2005.

[23] J. Bornholt, T. Mytkowicz, and K. McKinley. Uncertain<T>: A First-Order Type for Uncertain Data. *International Conference on Architectural Support for Programming Languages and Systems*, 2014.

[24] M. Bozzano and A. Villafiorita. Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform. In *Computer Safety, Reliability, and Security*, volume 2788 of *LNCS*, pages 49–62. Springer, 2003.

[25] M. Bozzano and A. Villafiorita. *Design and Safety Assessment of Critical Systems*. Auerbach Publications, 2010.

[26] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In *Automated Technology for Verification and Analysis*, volume 4762 of *LNCS*, pages 162–176. Springer, 2007.

[27] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Nguyen, T. Noll, and M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *Computer Safety, Reliability, and Security*, pages 173–186. Springer, 2009.

[28] M. Bozzano, A. Cimatti, M. Gario, and S. Tonetta. Formal Design of Fault Detection and Identification Components Using Temporal Epistemic Logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 326–340. Springer, 2014.

[29] M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal Safety Assessment via Contract-Based Design. In *Automated Technology for Verification and Analysis*, volume 8837 of *LNCS*, pages 81–97. Springer, 2014.

[30] M. Bozzano, A. Cimatti, A. Griggio, and C. Mattarei. Efficient Anytime Techniques for Model-Based Safety Analysis. In *Computer Aided Verification*, volume 9206 of *LNCS*, pages 603–621. Springer, 2015.

[31] M. Broy. Abstract Semantics of Synchronous Languages: The Example Esterel. Technical Report TUM-I9706, Technical University of Munich, 1997.

[32] D. Brugali and P. Scandurra. Component-Based Robotic Engineering (Part 1). *IEEE Robotics & Automation Magazine*, 16(4):84–96, 2009.

[33] D. Buede and W. Miller. *The Engineering Design of Systems: Models and Methods*. Wiley, 3rd edition, 2016.

[34] J. Butcher. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. Wiley, 2nd edition, 2003.

[35] D. Cancila, F. Terrier, F. Belmonte, H. Dubois, H. Espinoza, S. Gérard, and A. Cuccuru. SOPHIA: a Modeling Language for Model-Based Safety Engineering. In *Model-Based Architecting and Construction of Embedded Systems*. ACES-MB, 2009.

[36] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, 2010.

[37] J. Chaplin, O. Bakker, L. de Silva, D. Sanderson, E. Kelly, B. Logan, and S. Ratchev. Evolvable Assembly Systems: A Distributed Architecture for Intelligent Manufacturing. *Information Control Problems in Manufacturing*, 48(3):2065 – 2070, 2015.

[38] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An Open Source Tool for Symbolic Model Checking. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[39] A. Cimatti, I. Narasamdya, and M. Roveri. Software Model Checking SystemC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):774–787, 2013.

[40] D. Clarke, N. Diakov, R. Hähnle, E. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In *Formal Methods for Eternal Networked Software Systems*, pages 417–457. Springer, 2011.

[41] E. Clarke. The Birth of Model Checking. In *25 Years of Model Checking*, volume 5000 of *LNCS*, pages 1–26. Springer, 2008.

[42] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[43] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[44] E. Clarke, F. Lerda, and M. Talupur. An Abstraction Technique for Real-Time Verification. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 1–17. Springer, 2007.

[45] J. Cámara and R. de Lemos. Evaluation of Resilience in Self-Adaptive Systems Using Probabilistic Model-Checking. In *Software Engineering for Adaptive and Self-Managing Systems*, pages 53–62. IEEE, 2012.

[46] M. Cordy, A. Classen, P.-Y. Schobbens, P. Heymans, and A. Legay. Managing Evolution in Software Product Lines: A Model-Checking Perspective. In *Variability Modeling of Software-Intensive Systems*, pages 183–191. ACM, 2012.

[47] J. Curtis, K. Delaney, P. O'Kane, B. Roshto, and J. Sweeney. Hemodialysis Devices. In *Core Curriculum for the Dialysis Technician: A Comprehensive Review of Hemodialysis*. Medical Education Institute, 2008.

[48] R. de Lemos, H. Giese, H. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. Smith, J. Sousa, L. Tahvildari, K. Wong, and J. Wuttke. Software Engineering for Self-Adaptive Systems:

A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[49] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-state Verification. In *International Conference on Software Engineering*, pages 411–420. ACM, 1999.

[50] B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif. Towards Testing Self-organizing, Adaptive Systems. In *Testing Software and Systems*, pages 180–185. Springer, 2014.

[51] B. Eberhardinger, A. Habermaier, A. Hoffmann, A. Poeppel, and W. Reif. Toward Integrated Analysis & Testing of Component-Based, Adaptive Robot Systems. In *Verification and Validation of Adaptive Software Systems (to appear)*. IEEE, 2016.

[52] B. Eberhardinger, A. Habermaier, H. Seebach, and W. Reif. Back-to-Back Testing of Self-Organization Mechanisms. In *ICTSS 2016 (to appear)*. Springer, 2016.

[53] H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper. *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *LNCS*. Springer, 2004.

[54] E. Emerson. Model checking and the Mu-calculus. In *Series in Discrete Mathematics*, pages 185–214. American Mathematical Society, 1997.

[55] C. Ericson II. *Hazard Analysis Techniques for System Safety*. Wiley, 2005.

[56] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV: Overview and VerifyThis Competition. *International Journal on Software Tools for Technology Transfer*, 17(6):677–694, 2015.

[57] C. Esposito, D. Cotroneo, and N. Silva. Investigation on Safety-Related Standards for Critical Systems. In *Software Certification*, pages 49–54. IEEE, 2011.

[58] M. Esteve, J.-P. Katoen, V. Nguyen, B. Postma, and Y. Yushtein. Formal Correctness, Safety, Dependability, and Performance Analysis of a Satellite. In *International Conference on Software Engineering*, pages 1022–1031. IEEE, 2012.

[59] European Committee For Electrotechnical Standardization. EN 50128: Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems, 2011.

[60] P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[61] J.-D. Fekete, J. Wijk, J. Stasko, and C. North. The Value of Information Visualization. In *Information Visualization: Human-Centered Issues and Perspectives*. Springer, 2008.

[62] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements. In *Automated Software Engineering*, pages 283–292. IEEE, 2011.

[63] A. Filieri, G. Tamburrelli, and C. Ghezzi. Supporting Self-adaptation via Quantitative Verification and Sensitivity Analysis at Run Time. *IEEE Transactions on Software Engineering*, 1(99), 2015.

[64] C. Flanagan and J. Saxe. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *Principles of Programming Languages*, pages 193–205. ACM, 2001.

[65] Fondazione Bruno Kessler Embedded Systems Unit. *XSAP User Manual*. Fondazione Bruno Kessler, 2012.

[66] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[67] M. Güdemann. *Qualitative and Quantitative Formal Model-Based Safety Analysis*. PhD thesis, University of Magdeburg, 2011.

[68] M. Güdemann and F. Ortmeier. Model-Based Multi-objective Safety Optimization. In *Computer Safety, Reliability, and Security*, volume 6894 of *LNCS*, pages 423–436. Springer, 2011.

[69] S. Gerasimou, R. Calinescu, and A. Banks. Efficient Runtime Quantitative Verification Using Caching, Lookahead, and Nearly-optimal Reconfiguration. In *Software Engineering for Adaptive and Self-Managing Systems*, pages 115–124. ACM, 2014.

[70] H. Giese and M. Tichy. Component-Based Hazard Analysis: Optimal Designs, Product Lines, and Online-Reconfiguration. In *Computer Safety, Reliability, and Security*, pages 156–169. Springer, 2006.

[71] G. Gigante and D. Pascarella. Formal Methods in Avionic Software Certification: The DO-178C Perspective. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, volume 7610 of *LNCS*, pages 205–215. Springer, 2012.

[72] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification: Java SE 8 Edition*. Oracle, 2015.

[73] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[74] L. Grunske and B. Kaiser. An Automated Dependability Analysis Method for COTS-Based Systems. In *COTS-Based Software Systems*, pages 178–190. Springer, 2005.

[75] M. Güdemann, F. Ortmeier, and W. Reif. Safety and Dependability Analysis of Self-Adaptive Systems. In *Leveraging Applications of Formal Methods, Verification and Validation*, pages 177–184. IEEE, 2006.

[76] M. Güdemann, F. Ortmeier, and W. Reif. Using Deductive Cause-Consequence Analysis (DCCA) with SCADE. In *Computer Safety, Reliability, and Security*, pages 465–478. Springer, 2007.

[77] M. Güdemann, F. Nafz, F. Ortmeier, H. Seebach, and W. Reif. A Specification and Construction Paradigm for Organic Computing Systems. In *Self-Adaptive and Self-Organizing Systems*, pages 233–242, 2008.

[78] M. Güdemann, F. Ortmeier, and W. Reif. Computation of Ordered Minimal Critical Sets. In *Formal Methods for Automation and Safety in Railway and Automotives*, 2008.

[79] S. Gulan, S. Johr, R. Kretschmer, S. Rieger, and M. Ditze. Graphical Modelling meets Formal Methods. In *Industrial Informatics*, pages 716–721. IEEE, 2013.

[80] A. Habermaier, M. Güdemann, F. Ortmeier, W. Reif, and G. Schellhorn. The ForMoSA Approach to Qualitative and Quantitative Model-Based Safety Analysis. In *Railway Safety, Reliability, and Security*, pages 65–114. IGI Global, 2012.

[81] A. Habermaier, B. Eberhardinger, H. Seebach, J. Leupolz, and W. Reif. Runtime Model-Based Safety Analysis of Self-Organizing Systems with S#. In *Self-Adaptive and Self-Organizing Systems Workshops*, pages 128–133. IEEE, 2015.

[82] A. Habermaier, J. Leupolz, and W. Reif. Executable Specifications of Safety-Critical Systems with S#. In *Dependable Control of Discrete Systems*, pages 60–65. IFAC, 2015.

[83] A. Habermaier, A. Knapp, J. Leupolz, and W. Reif. Fault-Aware Modeling and Specification for Efficient Formal Safety Analysis. In *FMICS-AVoCS 2016 (to appear)*. Springer, 2016.

[84] A. Habermaier, J. Leupolz, and W. Reif. Unified Simulation, Visualization, and Formal Analysis of Safety-Critical Systems with S#. In *FMICS-AVoCS 2016 (to appear)*. Springer, 2016.

[85] R. Hähnle. The Abstract Behavioral Specification Language: A Tutorial Introduction. In *Formal Methods for Components and Objects*, pages 1–37. Springer, 2013.

[86] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[87] P. Hampton. Survey of Safety Architectural Patterns. In *Achieving Systems Safety: Safety-Critical Systems Symposium*, pages 137–158. Springer, 2012.

[88] L. Harms-Ringdahl. *Safety Analysis: Principles and Practice in Occupational Safety*. CRC Press, 2001.

[89] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal Analysis of the Remote Agent – Before and After Flight. In *The Fifth NASA Langley Formal Methods Workshop*, pages 716–721. NASA, 2000.

[90] A. Haxthausen. An Introduction to Formal Methods for the Development of Safety-critical Applications. Technical report, Technical University of Denmark, 2010.

[91] P. Helle. Automatic SysML-based Safety Analysis. In *Model Based Architecting and Construction of Embedded Systems*, pages 19–24. ACM, 2012.

[92] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[93] F. Hänsel, J. Poliak, R. Slovák, and E. Schnieder. Reference Case Study "Traffic Control Systems" for Comparison and Validation of Formal Specifications Using a Railway Model Demonstrator. In *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *LNCS*, pages 96–118. Springer, 2004.

[94] A. Hoffmann, A. Schierl, A. Angerer, M. Stüben, M. Vistein, and W. Reif. Robot Collision Avoidance Using an Environment Model for Capacitive Sensors. In *Planning, Control, and Sensing for Safe Human-Robot Interaction*. IEEE, 2015.

[95] C. Hoffmann, M. Sitharam, and B. Yuan. Making Constraint Solvers More Usable: Over-constraint Problem. *Computer-Aided Design*, 36(4):377–399, 2004.

[96] J. Hold and S. Perry. *SysML for Systems Engineering: A Model-Based Approach*. The Institution of Engineering and Technology, 2nd edition, 2013.

[97] F. Hölzl and M. Feilkas. AutoFocus 3 – A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 317–322. Springer, 2010.

[98] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[99] J. Huang. *Software Error Detection through Testing and Analysis*. Wiley, 2009.

[100] IAEA. The Fukushima Daiichi Accident: Report by the Director General. Technical report, International Atomic Energy Agency, 2015.

[101] Institute of Software and Systems Engineering. S# Repository. `https://github.com/isse-augsburg/ssharp`, 2016. Accessed: 2016-09-01.

[102] Institute of Software and Systems Engineering. S# Wiki. `https://github.com/isse-augsburg/ssharp/wiki`, 2016. Accessed: 2016-09-01.

[103] International Electrotechnical Commission. IEC 61511: Functional safety – Safety instrumented systems for the process industry sector, 2003.

[104] International Electrotechnical Commission. IEC 61508: Functional safety of electrical/-electronic/programmable electronic safety-related systems, 2010.

[105] International Organization for Standardization. ISO/IEC 23270: Information technology – Programming languages – C#, 2006.

[106] International Organization for Standardization. ISO 24765: Systems and software engineering – Vocabulary, 2010.

[107] International Organization for Standardization. ISO 26262: Road vehicles – Functional safety, 2011.

[108] International Organization for Standardization. ISO/IEC 9899: Information technology – Programming languages – C, 2011.

[109] International Organization for Standardization. ISO 10218: Robots and robotic devices – Safety requirements for industrial robots, 2011.

[110] International Organization for Standardization. ISO/IEC 23271: Information technology – Common Language Infrastructure (CLI), 2012.

[111] International Organization for Standardization. ISO/IEC 14882: Information technology – Programming languages – C++, 2014.

[112] E. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Object*, pages 142–164. Springer, 2012.

[113] A. Joshi, M. Whalen, and M. Heimdahl. Model-Based Safety Analysis Final Report. Technical report, NASA, 2006.

[114] B. Kaiser, P. Liggesmeyer, and O. Mäckel. A New Component Concept for Fault Trees. In *Safety Critical Systems and Software*, pages 37–46. Australian Computer Society, 2003.

[115] M. Käßmeyer, M. Schulze, and M. Schurius. A Process to Support a Systematic Change Impact Analysis of Variability and Safety in Automotive Functions. In *Software Product Line Conference*, pages 235–244. ACM, 2015.

[116] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *LNCS*, pages 692–707. Springer, 2015.

[117] J.-P. Katoen, M. Khattri, and I. Zapreevt. A Markov Reward Model Checker. In *Quantitative Evaluation of Systems*, pages 243–244. IEEE, 2005.

[118] Y. Kim, M. Kim, and T.-H. Kim. Statistical Model Checking for Safety Critical Hybrid Systems: An Empirical Evaluation. In *Hardware and Software: Verification and Testing*, pages 162–177. Springer, 2013.

[119] C. Kirsch and R. Sengupta. The Evolution of Real-Time Programming. In *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007.

[120] D. Klumpp, A. Habermaier, B. Eberhardinger, and H. Seebach. Optimising Runtime Safety Analysis Efficiency for Self-Organising Systems. In *Self-Adaptive and Self-Organizing Systems Workshops (to appear)*. IEEE, 2016.

[121] A. Knapp. Semantics of UML State Machines. Technical Report 0408, Institut für Informatik, Ludwig-Maximilians-Universität München, 2004.

[122] S. Kromodimoeljo and P. Lindsay. Automatic Generation of Minimal Cut Sets. In *Engineering Safety and Security Systems*, pages 33–47, 2015.

[123] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[124] A. Laarman. *Scalable Multi-Core Model Checking*. PhD thesis, University of Twente, 2014.

[125] C. Lee and S. Park. Survey on the Virtual Commissioning of Manufacturing Systems. *Journal of Computational Design and Engineering*, 1(3):213–222, 2014.

[126] A. Legay, B. Delahaye, and S. Bensalem. Statistical Model Checking: An Overview. In *Runtime Verification*, pages 122–135. Springer, 2010.

[127] K. Leino. Tools and Behavioral Abstraction: A Direction for Software Engineering. In *The Future of Software Engineering*, pages 115–124. Springer, 2011.

[128] F. Leitner-Fischer. *Causality Checking of Safety-Critical Software and Systems*. PhD thesis, University of Konstanz, 2015.

[129] F. Leitner-Fischer and S. Leue. QuantUM: Quantitative Safety Analysis of UML Models. In *Quantitative Aspects of Programming Languages*, volume 57 of *Electronic Proceedings in Theoretical Computer Science*, pages 16–30. Open Publishing Association, 2011.

[130] J. Leupolz, A. Habermaier, and W. Reif. Safety Analysis of a Hemodialysis Machine with S#. In *EuroAsiaSPI² (to appear)*. Springer, 2016.

[131] N. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2011.

[132] N. Leveson and C. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7): 18–41, 1993.

[133] O. Lichtenstein, A. Pnueli, and L. Zuck. The Glory of The Past. In *Logics of Programs*, pages 196–218. Springer, 1985.

[134] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain. In *Model-Driven Engineering Languages and Systems*, pages 166–182. Springer, 2014.

[135] M. Lipaczewski, S. Struck, and F. Ortmeier. Using Tool-Supported Model Based Safety Analysis – Progress and Experiences in SAML Development. In *High-Assurance Systems Engineering*, pages 159–166. IEEE, 2012.

[136] E. Lippert. C++ and the Pit Of Despair. `https://blogs.msdn.microsoft.com/ericlippert/2007/08/14/c-and-the-pit-of-despair/`, 2007. Accessed: 2016-06-10.

[137] B. Liptak. *Instrument Engineers' Handbook, Volume Two: Process Control and Optimization*. CRC Press, 4th edition, 2005.

[138] O. Lisagor, M. Bozzano, M. Bretschneider, and T. Kelly. Incremental Safety Assessment: Enabling the Comparison of Safety Analysis Results. In *International System Safety Conference*. System Safety Society, 2010.

[139] N. Markey. Temporal Logic with Past is Exponentially More Succinct. In *EATCS Bulletin*, volume 79, pages 122–128. European Association for Theoretical Computer Science, 2003.

[140] A. Mashkoor. The Hemodialysis Machine Case Study. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 329–343. Springer, 2016.

[141] J. Meseguer. Twenty Years of Rewriting Logic. In *Rewriting Logic and Its Applications*, pages 15–17. Springer, 2010.

[142] F. Mhenni, N. Nguyen, H. Kadima, and J. Choley. Safety Analysis Integration in a SysML-Based Complex System Design Process. In *Systems Conference*, pages 70–75. IEEE, 2013.

[143] F. Mhenni, N. Nguyen, and J. Choley. Automatic Fault Tree Generation from SysML System Models. In *Advanced Intelligent Mechatronics*, pages 715–720. IEEE, 2014.

[144] F. Mhenni, J. Choley, and N. Nguyen. SysML Extensions for Safety-Critical Mechatronic Systems Design. In *Systems Engineering*, pages 242–247. IEEE, 2015.

[145] Modelica Association. *Modelica – A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.3*, 2014.

[146] F. Nafz, J.-P. Steghöfer, H. Seebach, and W. Reif. Formal Modeling and Verification of Self-* Systems Based on Observer/Controller-Architectures. In *Assurances for Self-Adaptive Systems*, pages 80–111. Springer, 2013.

[147] A. Nathan. *WPF 4.5 Unleashed*. Sams Publishing, 2013.

[148] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[149] K. Ng, M. Warren, P. Golde, and A. Hejlsberg. The Roslyn Project: Exposing the C# and VB compiler's code analysis. *White Paper, Microsoft*, 2012.

[150] H. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley and Sons, 1999.

[151] T. Noll. Safety, Dependability and Performance Analysis of Aerospace Systems. In *Formal Techniques for Safety-Critical Systems*, volume 476 of *CCIS*, pages 17–31. Springer, 2015.

[152] NUnit.org. NUnit Documentation Wiki. `https://github.com/nunit/docs/wiki`, 2016. Accessed: 2016-09-01.

[153] Object Management Group. OMG Systems Modeling Language (OMG SysML), Version 1.4, 2015.

[154] Object Management Group. OMG Unified Modeling Language (OMG UML), Version 2.5, 2015.

[155] F. Ortmeier and W. Reif. Failure-Sensitive Specification. Technical Report 2004-3, University of Augsburg, 2004.

[156] F. Ortmeier and W. Reif. Safety Optimization: A combination of fault tree analysis and optimization techniques. In *Dependable Systems and Networks*, pages 651–658, 2004.

[157] F. Ortmeier and G. Schellhorn. Formal Fault Tree Analysis – Practical Experiences. *Electronic Notes in Theoretical Computer Science*, 185:139–151, 2007.

[158] F. Ortmeier, G. Schellhorn, A. Thums, W. Reif, B. Hering, and H. Trappschuh. Safety Analysis of the Height Control System for the Elbtunnel. In *Computer Safety, Reliability and Security*, volume 2434 of *LNCS*, pages 296–308. Springer, 2002.

[159] F. Ortmeier, W. Reif, and G. Schellhorn. Deductive Cause-Consequence Analysis (DCCA). In *IFAC World Congress*. Elsevier, 2006.

[160] F. Ortmeier, M. Güdemann, and W. Reif. Formal Failure Models. In *Dependable Control of Discrete Systems*, pages 145–150. IFAC, 2007.

[161] C. Otero. *Software Engineering Design: Theory and Practice*. CRC Press, 2012.

[162] R. Paige, L. Rose, X. Ge, D. Kolovos, and P. Brooke. FPTC: Automated Safety Analysis for Domain-Specific Languages. In *Models in Software Engineering*, pages 229–242. Springer, 2009.

[163] Y. Papadopoulos, M. Walker, D. Parker, E. Rüde, R. Hamann, A. Uhlig, U. Grätz, and R. Lien. Engineering Failure Analysis and Design Optimisation with HiP-HOPS. *Engineering Failure Analysis*, 18(2):590 – 608, 2011.

[164] D. Peled. Ten Years of Partial Order Reduction. In *Computer Aided Verification*, pages 17–28. Springer, 1998.

[165] M. Pelikan, D. Goldberg, and E. Cantú-Paz. Hierarchical Problem Solving and the Bayesian Optimization Algorithm. In *Genetic and Evolutionary Computation Conference*, pages 267–274. Morgan Kaufmann, 2000.

[166] H. Pentti and H. Atte. Failure Mode and Effects Analysis of Software-Based Automation Systems. In *VTT Industrial Systems, STUK-YTO-TR 190*, page 190, 2002.

[167] Personalized Medicine Coalition. The Case for Personalized Medicine – 4th Edition, 2014.

[168] G. Plotkin. *A Structural Approach to Operational Semantics*. University of Edinburgh, 2004.

[169] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *Theoretical Aspects of Computer Software*, pages 244–264. Springer, 1991.

[170] M. Pradella, P. San Pietro, P. Spoletini, and A. Morzenti. Practical Model Checking of LTL with Past. In *Automated Technology for Verification and Analysis*, 2003.

[171] D. Prasanna. *Dependency Injection*. Manning Publications, 2009.

[172] C. Preschern, N. Kajtazovic, and C. Kreiner. Building a Safety Architecture Pattern System. In *European Conference on Pattern Languages of Programs*, pages 17:1–17:55. ACM, 2015.

[173] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[174] Radio Technical Commission for Aeronautics. DO-178C: Software Considerations in Airborne Systems and Equipment Certification, 2011.

[175] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck. Towards a generic observer/controller architecture for Organic Computing. *GI Jahrestagung*, 93:112–119, 2006.

[176] Roslyn Team, Microsoft. Language Feature Status. `https://github.com/dotnet/roslyn/blob/master/docs/Language%20Feature%20Status.md`, 2016. Accessed: 2016-08-05.

[177] I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *Feature-Oriented Software Development*, pages 49–56. ACM, 2010.

[178] A. Schiendorfer, J.-P. Steghöfer, A. Knapp, F. Nafz, and W. Reif. Constraint Relationships for Soft Constraints. In *Research and Development in Intelligent Systems*, pages 241–255. Springer, 2013.

[179] K. Schneider. The Synchronous Programming Language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, 2010.

[180] H. Seebach, F. Nafz, J.-P. Steghöfer, and W. Reif. How to Design and Implement Self-organising Resource-Flow Systems. In *Organic Computing — A Paradigm Shift for Complex Systems*, pages 145–161. Springer, 2011.

[181] S. Sharvia and Y. Papadopoulos. Integrating Model Checking with HiP-HOPS in Model-Based Safety Analysis. *Reliability Engineering & System Safety*, 135:64–80, 2015.

[182] D. Smith and K. Simpson. *Safety Critical Systems Handbook: A straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849*. Elsevier, 2010.

[183] Society of Automotive Engineers. AS5506B: Architecture Analysis & Design Language (AADL), 2012.

[184] J.-P. Steghöfer. *Large-Scale Open Self-Organising Systems: Managing Complexity with Hierarchies, Monitoring, Adaptation, and Principled Design*. PhD thesis, University of Augsburg, 2014.

[185] J.-P. Steghöfer, P. Mandrekar, F. Nafz, H. Seebach, and W. Reif. On Deadlocks and Fairness in Self-organizing Resource-Flow Systems. In *Architecture of Computing Systems*, pages 87–100. Springer, 2010.

[186] N. Storey. *Safety Critical Computer Systems*. Addison-Wesley, 1996.

[187] S. Struck, M. Lipaczewski, F. Ortmeier, and M. Güdemann. Multi-objective Optimization of Formal Specifications. In *High-Assurance Systems Engineering*, pages 201–208. IEEE, 2012.

[188] S. Struck, M. Güdemann, and F. Ortmeier. Efficient Optimization of Large Probabilistic Models. *Journal of Systems and Software*, 86(10):2488–2501, 2013.

[189] O. Tardieu. A Deterministic Logical Semantics for Pure Esterel. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.

[190] S. Thiel, S. Ferber, T. Fischer, A. Hein, and M. Schlick. A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems. Technical Report 2001-01-0025, Society of Automotive Engineers, 2001.

[191] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, 2014.

[192] M. Trapp and D. Schneider. Safety Assurance of Open Adaptive Systems – A Survey. In *Models@run.time*, pages 279–318. Springer, 2014.

[193] W. Vesely, J. Dugan, J. Fragola, J. Minarick, and J. Railsback. Fault Tree Handbook with Aerospace Applications. Technical report, NASA, 2002.

[194] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. In *Automated Software Engineering*, pages 203–232. Springer, 2003.

[195] W. Visser, M. Dwyer, and M. Whalen. The Hidden Models of Model Checking. *Software & Systems Modeling*, 11(4):541–555, 2012.

[196] M. Vistein. *Embedding Real-Time Critical Robotics Applications in an Object-Oriented Language*. PhD thesis, University of Augsburg, 2015.

[197] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif. Flexible and Continuous Execution of Real-Time Critical Robotic Tasks. *International Journal of Mechatronics and Automation*, 4(1), 2014.

[198] M. Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013.

[199] M. Vouk. Back-to-Back Testing. In *Information and Software Technology*, pages 34 – 45. Elsevier, 1990.

[200] A. West. NASA Study on Flight Software Complexity. Technical report, NASA, 2009.

[201] D. Weyns. Towards an Integrated Approach for Validating Qualities of Self-adaptive Systems. In *Workshop on Dynamic Analysis*, pages 24–29. ACM, 2012.

[202] D. Weyns, M. Iftikhar, D. de la Iglesia, and T. Ahmad. A Survey of Formal Methods in Self-Adaptive Systems. In *International C* Conference on Computer Science and Software Engineering*, pages 67–79. ACM, 2012.

[203] M. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. Heimdahl, and S. Rayadurgam. Your "What" Is My "How": Iteration and Hierarchy in System Design. *IEEE Software*, 30(2): 54–60, 2013.

[204] R. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann, 2003.

[205] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

[206] J. Woodcock, P. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computation Survey*, 41(4):19:1–19:36, 2009.

[207] E. Yip, M. Kuo, P. Roop, and D. Broman. Relaxing the Synchronous Approach for Mixed-Criticality Systems. In *Real-Time and Embedded Technology and Applications Symposium*, pages 89–100. IEEE, 2014.

Symbols

	Domains and Meta-Variables
K	a Kripke structure
M	an executable model
$p \in P$	a proposition of a Kripke structure or executable model
$\vartheta \in P$	a RIA predicate
$\varrho \in P$	a flag signaling that no further reconfigurations are possible
$f \in F$	a fault of a fault-aware Kripke structure or executable model
$\Gamma, \Delta \subseteq F$	subsets of faults
Λ_H	a set of minimal critical fault sets for hazard H
$s \in S$	a state of a Kripke structure
R	a transition relation of a Kripke structure
L	a labeling function of a Kripke structure
I	a set of initial states of a Kripke structure or the initialization program of an executable model
$E \in \mathit{Prog}$	an execution program of an executable model
$v \in \mathit{Val}$	a literal value of Boolean or integer type
$x \in V$	a variable of Boolean or integer type
$\sigma \in \Sigma$	a variable environment of a formal program
$e \in \mathit{Expr}$	an expression of a formal program
$\rho \in \mathit{Prog}$	a formal program
ς	a path fragment of a Kripke structure
$\varphi \in \Phi$	an LTL formula
$\psi \in \Psi \subseteq \Phi$	a persistency constraint formulated in LTL
$H \in \Phi$	a propositional logic formula specifying a hazard

Functions and Operators

$K_1 \equiv_F K_2$	path-equivalent Kripke structures modulo faults
$K \triangleleft F$	a set of extended Kripke structures after fault injection
$K \setminus\!\setminus F$	a set of reduced Kripke structures after fault removal
$K(M)$	an induced fault-aware Kripke structure for an executable model
$\mathcal{E}(K)$	a fault-enforced fault-aware Kripke structure
$\varsigma_{\mathrm{S}}[n]$	the selection of the n-th state in path ς
$\varsigma_{\mathrm{F}}[n]$	the selection of the n-th set of activated faults in path ς
$paths(K)$	the set of all paths of a Kripke structure
$\mathcal{R}(\cdot)$	the set of reachable states of a Kripke structure K or a path ς
$\mathcal{A}(\cdot)$	the set of activatable faults of a Kripke structure K or a path ς
act-min$(\cdot)$	the activation minimization function used by formal programs
$f_1 \preceq f_2$	a relation between faults indicating that f_1 subsumes f_2
$\mathcal{S}(\Gamma)$	the set of faults subsumed by fault set Γ
$\mathcal{E}_\Gamma[\![\cdot]\!]$	the expression semantics of formal programs
$\mathcal{P}_\Gamma[\![\cdot]\!]$	the semantics of formal programs