

Runtime model-based safety analysis of self-organizing systems with S

Axel Habermaier, Benedikt Eberhardinger, Hella Seebach, Johannes Leupolz, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Habermaier, Axel, Benedikt Eberhardinger, Hella Seebach, Johannes Leupolz, and Wolfgang Reif. 2015. "Runtime model-based safety analysis of self-organizing systems with S#." In 2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, 21-25 September 2015, Cambridge, MA, USA, edited by Gerrit Anders, Jean Botev, and Markus Esch, 128-33. Piscataway, NJ: IEEE. <https://doi.org/10.1109/sasow.2015.26>.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/>



Runtime Model-Based Safety Analysis of Self-Organizing Systems with S#

Axel Habermaier, Benedikt Eberhardinger, Hella Seebach, Johannes Leupolz, and Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg, Germany
E-Mail: {habermaier, eberhardinger, seebach, leupolz, reif}@isse.de

Abstract—Self-organizing systems present a challenge for model-based safety analysis techniques: At design time, the potential system configurations are unknown, making it necessary to postpone the safety analyses to runtime. At runtime, however, model checking based safety analysis techniques are often too time-consuming because of the large state spaces that have to be analyzed. Based on the S# framework’s support for runtime model adaptation, we modularize runtime safety analyses by splitting them into two parts, modeling and analyzing the self-organizing and non-self-organizing parts separately. With some additional heuristics, the resulting state space reduction facilitates the use of model checking based safety analysis techniques to analyze the safety of self-organizing systems. We outline this approach on a self-organizing production cell, assessing the self-organization’s impact on the overall safety of the system.

Keywords—safety analysis, formal methods, model checking, self-organizing systems, models at runtime

I. INTRODUCTION

Model-based safety analysis techniques are able to automatically compute all *minimal cut sets* of a system [1]. These sets represent combinations of component faults that can potentially cause the occurrences of *safety hazards*, i.e., situations in which a system can cause environmental damage, injuries, or loss of lives. Safety-critical self-organizing systems dynamically adapt their behavior and structure to changes in their environment, in particular to occurrences of component faults, resulting in system configurations that often cannot be predicted at design time. It is therefore necessary to postpone model-based safety analyses to runtime when more information about the actual configurations of the systems is available. Runtime adaptation of the models in accordance with the systems’ self-organization [2] enables the use of model-based safety analysis techniques for self-organizing systems, but the large state spaces of these models make the analyses time-consuming. Consequently, fully automated runtime safety analyses of self-organizing systems with integrated tool support are still an open research question [3]–[6].

This position paper introduces a systematic model-based safety analysis approach for self-organizing systems that is conducted at runtime. It is based on our S# modeling and analysis framework, originally envisioned for design time safety analyses of non-self-organizing systems [7]. We outline how S#’s modular and configurable models also allow for formal safety analyses at runtime, sharing the development and runtime models of the systems [8]. We discuss some ideas to lessen the combinatorial state space explosion problem when analyzing self-organizing systems using our model checking based safety analysis technique *Deductive Cause-Consequence*

Analysis (DCCA) [1]. In particular, we propose to classify component faults as either *tolerable* or *intolerable* [9], depending on whether the system’s self-organization mechanism is able to compensate their occurrences. Based on this classification, we modularize the safety analyses by examining tolerable and intolerable faults separately, with the combination of the individual results being equivalent to a non-modularized DCCA. The resulting reduction in analysis times allows us to conduct safety analyses of self-organizing systems at runtime. Additionally, our approach can also be used to identify the limits of self-organization mechanisms, which usually cannot be designed in a way such that all possible safety hazards are prevented. We outline all of these ideas using a well-known case study: a self-organizing production cell [10].

The paper is organized as follows: The next section introduces the case study, followed by Section III with an overview of our S# framework for modeling and analyzing safety-critical systems. Sections IV and V outline how S# can be used to model and analyze the safety aspects of self-organizing systems and the case study in particular. We consider related work in Section VI and conclude with Section VII, briefly discussing some ideas for future work.

II. THE SELF-ORGANIZING PRODUCTION CELL

Our running example is a self-organizing production cell consisting of robots and carts [10]. The carts transport workpieces between the robots, which have several switchable tools such as drills and screwdrivers that they use on the workpieces. Figure 1 shows a simple configuration of a production cell consisting of three robots and two carts connecting them. The robots and carts are responsible for processing incoming workpieces in a given sequence of tool applications; in our case study, the processing sequence is to drill a hole, insert a screw, and then tighten the screw. Robot R_1 , for instance, is responsible for drilling a hole into a workpiece and transferring it to cart C_1 afterwards. The cart then transports the workpiece to robot R_2 , which inserts the screw into the previously drilled hole, and so on.

The production cell is self-organizing as it can reconfigure itself to compensate for broken tools or to incorporate new tools, robots, or carts, for instance. Such reconfigurations are initiated and coordinated either by a central *observer/controller* [11] or by the leader of a local *coalition* [12], which determine new paths for the carts as well as the set of tools to be used by the individual robots. The central observer/controller stops the entire system in order to calculate new configurations and to assign them to the robots and carts. Coalitions, on the other hand, are (potentially small) groups

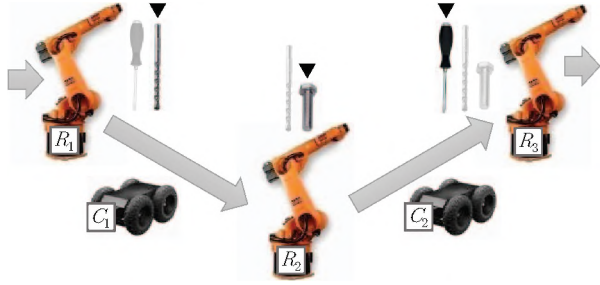


Figure 1. A simple configuration of the self-organizing production cell with three robots and two carts connecting them. Adjacent to each robot are the tools it can use, with the currently active one being marked by an arrow head.

of robots and carts that can locally react to the event or situation that triggered the reconfiguration. Once the coalition has been formed, its leader acts like a local observer/controller, using a *constraint solver* or a *genetic algorithm* to compute a new configuration; note that the latter might fail to find any valid configurations even though there actually are some. Subsequently, the newly computed configurations are checked by a *result checker* to ensure their validity.

In this paper, we do not consider the inner workings of the robots and the carts. In particular, we are not interested in trajectory planning for the robots, the robots' tool switching mechanisms, or the carts' position tracking system. Intolerable component faults that we intend to analyze are: Sensors do not detect the arrival of a workpiece, carts move to the wrong robot, and robots and carts receive no reconfiguration information due to, for instance, networking problems. Tolerable component faults, on the other hand, are broken tools, robots that can no longer switch tools, robots that are no longer working at all, and carts whose assigned paths are blocked.

III. OVERVIEW OF S#

S# is an integrated, tool-supported approach for modeling and analyzing safety-critical systems. S# models are *executable*, allowing them to be simulated, tested, visualized, and debugged in addition to using formal analysis techniques [7]. Safety analyses make use of our model checking based DCCA to rigorously assess the safety of a system by automatically computing the minimal cut sets of the safety hazards [1]. The underlying model of computation is a series of discrete system steps, where each step takes the same amount of time. Structural and behavioral design variants can be modeled using the modularity and composability concepts of S#'s modeling language, which is most useful when exploring the design space of safety-critical systems early in their development or when developing safety-critical product lines [13].

A. The S# Modeling Language

S# provides a component-oriented *domain specific language* embedded into the C# programming language. In other words, S# models are *represented* as C# programs; conceptually, however, these programs are still models of the safety-critical systems to be analyzed. In particular, some parts of those programs 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 S# models that do in fact represent software components

are *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.

S# inherits all of C#'s language features and expressiveness. Every .NET library and tool can be used, including all state-of-the-art code editing and debugging features provided by the Visual Studio development environment. However, some restrictions apply to those parts of a S# model that are to be model checked: Recursion, loops, arbitrary object construction, amongst others, are disallowed. The remaining parts of the models are completely unrestricted, including, for instance, the code that adapts a model at runtime in accordance with a reconfiguration of a self-organizing system.

B. Systematic Modeling of Safety-Critical Systems

Adequate models of the systems to be analyzed are a prerequisite for any model-based safety analysis technique. Consequently, S# provides specifically tailored modeling language concepts as well as a systematic modeling methodology that help to improve the adequacy of the models. In particular, S# requires the modeling of relevant component *faults* and the *physical environment* of safety-critical systems. The model of the *intended* system behavior must be extended with the *occurrence patterns* of faults as well as their *local effects* on the affected components. Subsequently, formal analysis techniques are able to reason about the system's *global* behavior in degraded situations. The physical environment of a system represents those parts of the physical world that are indirectly influenced or observed by the system (usually via actuators and sensors), such as the workpieces of the production cell. The physical environment is required for the correct specification of safety hazards, as hazards typically result from a discrepancy between the physical environment's actual state and its state as it is perceived by the system. In turn, these discrepancies occur either because of systematic design errors, that is, the system is functionally incorrect, or because of occurrences of one or more component faults.

When modeling safety-critical systems, S#'s modeling methodology suggests to first identify the physical environment that the system is intended to indirectly observe and influence. In particular, it must be possible to express the safety hazards that are to be analyzed (typically exclusively) in terms of the physical environment model. The next step consists of determining the sensors and actuators that are available to the system. Only then should the actual controller components of the system be modeled, which can be comprised of multiple subsystems of various kinds such as software components, hydraulic components, pneumatic components, etc. Component faults are either modeled together with the individual components or they are added later in a separate step.

Figure 2 shows an overview of the components constituting the model of the case study. The case study's physical environment consists of the workpieces that are modified by the tools attached to the robots. Consequently, the tools take on the roles of actuators, just like the carts are actuators that can move the workpieces around. Each robot has a sensor that detects whether a workpiece is positioned in front of it, allowing the robot to use its tool on the workpiece. The model

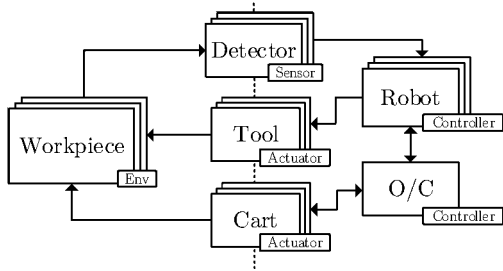


Figure 2. Overview of the components constituting the model of the case study. Each component represents some part of the physical environment, a sensor, an actuator, or a controller. The connections between the components indicate the information flow within the model. The dashed line in the center separates the physical environment on the left from the system’s controller components on the right. The system’s sensors and actuators are drawn on top of the line because they connect the two parts of the model.

contains multiple workpieces, robots, sensors, tools, and carts, but there is only one observer/controller (O/C) that monitors the carts and robots for faults, trying to reorganize the system to work around their occurrences. The distinction between local and global reconfiguration mechanisms is irrelevant for the model, as the safety analyses only have to distinguish between successful and failed reconfigurations.

C. Deductive Cause-Consequence Analysis

DCCA is S#’s fully automated, model checking based safety analysis technique. For a safety hazard H , DCCA computes all minimal cut sets by individually checking all combinations of component faults Γ , determining whether such a set Γ does or does not have the potential to cause an occurrence of H . For instance, the fault of cart C_1 in Fig. 1 that causes it to transport a workpiece to robot R_3 instead of R_2 is a minimal cut set for the hazard of damaging a workpiece, as the use of the screwdriver before a screw has been inserted might result in scratches. Formally, for a system model S with a set of component faults Δ , S# uses a *Computation Tree Logic* formula to check whether $\Gamma \subseteq \Delta$ is a cut set for H [1], [14]:

Definition (Minimal Cut Sets). A set of component faults $\Gamma \subseteq \Delta$ is a *cut set* for hazard H iff $S \models \text{only}_\Delta(\Gamma) \text{ EU } H$, where $\text{only}_\Delta(\Gamma) := \bigwedge_{\delta \in \Delta \setminus \Gamma} \neg \delta$. A cut set Γ is *minimal* if no proper subset $\Gamma' \subset \Gamma$ is a cut set.

The formula characterizes a cause-consequence relationship between the component faults (the causes) and the hazard (the consequence): A set of component faults Γ is a cut set for a hazard H if and only if there is the possibility that H occurs and before that, at most the faults in Γ have occurred. DCCA has exponential complexity as it has to check all combinations of component faults. In practice, however, the number of required checks usually is significantly lower, as the cut set property is *monotonic* with respect to set inclusion, that is, once a set of component faults Γ is known to be a cut set, all supersets Γ' with $\Gamma \subseteq \Gamma'$ are cut sets as well.

IV. MODELING SELF-ORGANIZING SYSTEMS WITH S#

In S#, components of safety-critical systems can be modeled with different levels of abstraction. During the early phases of development as well as for model-based safety analysis purposes, the abstraction level is typically rather high.

Subsequently, the abstract component behavior and system structure can be refined to include more details for fine-grained simulation-only models, or different parts of the more detailed model can be extracted and formally analyzed individually. In particular, safety analyses are often decomposed such that off-the-shelf components are not analyzed in detail: For these components, the respective vendors already determined the 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]. This is a time-tested practice with a long tradition in *Fault Tree Analysis*, for instance [15]. Model-based safety analysis techniques can make use of the same principles to reduce the complexities and state spaces of the models, focusing on the safety hazards that result from the *composition* of the components rather than getting lost in the details of the fault behaviors of the individual components.

We extend this decomposition approach to self-organizing systems in order to modularize both the models as well as the safety analyses [16]: We manually separate *tolerable* faults (such as broken tools) from *intolerable* ones (such as failures of workpiece sensors), depending on whether the system can tolerate their occurrences due to its self-organization mechanism. Tolerable faults are those faults the system can compensate, continuing correct and safe operation after a reconfiguration. Self-organization can therefore be seen as a safety mechanism that increases the system’s *fault tolerance* in order to prevent safety hazards for as long as possible [9]. However, self-organization cannot cope with all faults that occur during the lifetime of a system: Intolerable faults are outside of its reach, either because their occurrences cannot be detected or there is no possible way to react to their occurrence. In particular, a fault discovery mechanism might be missing due to a deliberate design decision in order to reduce costs or because the discovery is physically impossible for some reason. Additionally, at some point, there is not enough redundancy left in the system to continue operating safely after the occurrence of a tolerable fault, in which case a *reconfiguration failure* occurs. In the case study, reconfiguration fails, for instance, when all tools of the same kind no longer work.

A. Modeling the Case Study

Listing 1 shows a partial S# component representing a robot of the production cell. S# component *types* are represented by C# classes derived from `Component`, whereas S# component *instances* are represented by .NET objects of the corresponding C# class. *Provided* and *required ports* of the components are represented by methods, with required ports having no body and being marked with the `extern` keyword (not shown in Listing 1). The `Robot` component is configurable as the tools it has available are set via the constructor, allowing the creation of `Robot` instances with different sets of tools. The observer/controller uses the provided port `Reconfigure` to set the sequence of tools the robot should use, with the `_currentTool` field indicating which tool is currently being used by the robot.

In S#, faults are modeled by adding nested classes derived from `Fault` to a component, using attributes to specify the *occurrence patterns* of the faults. In Listing 1, the `MissedReconfiguration` fault is *transient*, indicating that

```

class Robot : Component {
  Tool[] _tools; int[] _toolsToUse; int _currentTool;

  public Robot(Tool[] tools) { _tools = tools; /* ... */ }

  private void UseCurrentTool()
  { _tools[_toolsToUse[_currentTool]].Use(); }

  private void SwitchToNextTool() { ++_currentTool; }

  public void Reconfigure(int[] toolsToUse)
  { _toolsToUse = toolsToUse; _currentTool = 0; }

  [Transient] class MissedReconfiguration : Fault {
    public void Reconfigure(int[] toolsToUse) { }
  }

  [Persistent] class CannotUseAnyTools : Fault {
    public void UseCurrentTool() { }
  }

  [Persistent] class CannotSwitchTools : Fault {
    public void SwitchToNextTool() { }
  }

  /* ... */
}

```

Listing 1. Partial S# component representing a robot of the production cell, showing some of the internal state, three provided ports, and three of the robot’s faults. Other internal state, the subcomponents, and the state machine describing the robot’s behavior are omitted due to space restrictions.

the fault randomly occurs for some time, whereas the other two faults are *persistent*, i.e., once they occur, they never disappear again. The faults’ methods represent their *fault effects*; for instance, the fault effect of the `MissedReconfiguration` fault replaces the implementation of the component’s provided port `Reconfigure` with a do-nothing instruction whenever the fault occurs [1], [7].

B. Abstract Specification of the Observer/Controller

We closely follow the approach outlined by Gdemann to specify the behavior of the observer/controller [17]. Nafz et al. formally verified the correctness of the result checker [10], allowing us to assume that the observer/controller either returns a *valid* configuration or none at all. We abstractly specify this behavior by allowing the model of the observer/controller to assign *any* configuration; we then filter out *invalid* ones by instructing the model checker to ignore all traces with invalid configurations when performing the DCCA. Such a specification of the observer/controller supports all kinds of reconfiguration mechanisms, ranging from global constraint solving approaches to local coalition formations.

V. ANALYZING SELF-ORGANIZING SYSTEMS WITH S#

S#’s support for parameterized component instantiation and model adaptation allows for runtime safety analyses of different configurations of self-organizing systems. In particular, it is possible to instantiate only certain parts of a model in order to check the limits of the reconfiguration mechanisms independently from the overall system safety. This decomposition approach reduces the combinatorial state space explosion problem: The reachable state space of a model of a self-organizing system is already very large because of the high redundancy required for fault tolerance. Additional states are introduced by the occurrence patterns of faults, as S# has to add at least one Boolean variable to the model for each fault. Therefore, the state space increases by a factor of 2^n when a set of component faults with cardinality n is checked, which obviously does not scale.

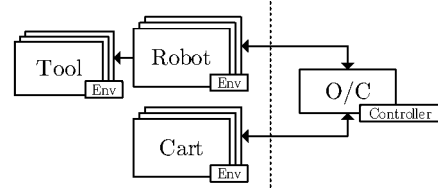


Figure 3. Overview of the pruned model focusing on the production cell’s local or global self-organization mechanism. All states introduced by the workpieces and the workpiece detectors are gone and S# or the model checker can optimize out most of the now unreachable states of the robots and carts.

All safety-critical systems suffer from this problem, but it is even worse for self-organizing ones: Typically, non-self-organizing systems have rather small minimal cut sets, often containing only very few faults. Even if there are larger ones, they might not have to be computed explicitly, as the contribution to the overall occurrence probability of a hazard decreases the more faults a cut set contains [15]. For self-organizing systems, on the other hand, minimal cut sets containing tolerable faults become quite large, yet we have to compute them anyway in order to determine the limits of the reconfiguration mechanism. We mitigate the situation by separating the safety analyses for tolerable and intolerable faults into two parts, using runtime model adaptation and some additional heuristics to conduct the analyses more efficiently.

A. Analyzing Tolerable Faults

We analyze the tolerable faults of a self-organizing system to determine the limits of its self-organization mechanism, that is, we compute the minimal cut sets of tolerable faults that prevent the system from further self-organization. The minimal cut sets have to be recomputed after each reconfiguration of the system, as they can change in various ways when, for instance, robots are removed from or added to the system. The analyses exclusively focus on the self-organizing aspects of the system model, considerably reducing its state space. For the case study, for instance, we take the point of view that the robots, the tools, and the carts constitute the physical environment that the observer/controller monitors and influences. Figure 3 gives an overview of such a pruned model consisting of the aforementioned components only; while the components of the full model can be reused, the instantiation of the pruned model has to be performed manually. Yet there are still two sources of exponential complexity: DCCA has to check all combinations of tolerable faults and the model checker has to enumerate all possible configurations every time some tolerable faults trigger a reconfiguration. In other words, we have an exponential amount of formulas to check on a model that still includes the occurrence states of all analyzed faults and that branches exponentially after every reconfiguration.

To lessen the first problem, we no longer check for cut sets by increasing cardinality, instead relying on a *binary search* with respect to the size of the potential cut sets; that way, larger minimal cut sets are found more quickly in the average case. The monotonicity of the cut set property then allows us to efficiently rule out large numbers of potential cut sets, as we no longer have to check those explicitly: If a set of component faults Γ is a cut set, we do not have to check any supersets that contain Γ , whereas if Γ is not a cut set, we do not have to check any of its subsets.

To mitigate the second problem, we exploit some additional knowledge about the way the reconfiguration mechanism is modeled: Only the occurrences of tolerable faults trigger reconfigurations and we only want to determine whether, given a set of tolerable faults, there still is a valid successor configuration. Consequently, S# can automatically adapt the model so that the analyzed faults Γ already occur in the initial state and never disappear, effectively making all tolerable faults in Γ persistently occurring. As a result, the reachable state space of the adapted model is reduced by a factor of $2^{|\Gamma|}$, as the occurrence states of those faults no longer have to be tracked explicitly; they are built into the model. Assuming a system configuration as the one shown in Fig. 1, for example, the system can no longer reconfigure itself once both screwdrivers are broken. It is irrelevant, however, if R_1 's or R_3 's screwdriver breaks first, or whether R_2 's drill, for instance, breaks in-between as well. Consequently, when S# adapts the case study model such that both screwdrivers are persistently broken, the model checker can determine more efficiently that these two faults represent a cut set. Additionally, we can simplify the DCCA formula to **EXH** for the adapted models, checking only whether the *next* reconfiguration can still find a valid successor configuration. The model's built-in tolerable faults Γ are a cut set if and only if the formula holds.

With these two ideas, we now only check, on average, a non-exponential number of adapted models with further reduced state spaces, and each check only considers one reconfiguration instead of possibly many. Our approach therefore reduces the runtime of a DCCA; however, we have not yet systematically evaluated the achieved speedups. Additionally, the longer a system is running, the more tolerable faults have already occurred, further decreasing the analysis times as the numbers and sizes of cut set candidates decrease as well. Over time, however, the numbers and sizes of the minimal cut sets can also increase again, namely when new robots are added to the system or broken tools are replaced, for instance.

B. Analyzing Intolerable Faults

To assess the overall safety of a self-organizing system such as the production cell, we have to compute the minimal cut sets containing the system's intolerable faults. We define a general reconfiguration failure that subsumes all tolerable faults and accounts for the fact that the reconfiguration mechanism might be unable to find a valid configuration even though one still exists. The minimal cut sets therefore become smaller and can be found more quickly as they contain at most the reconfiguration failure, but no tolerable faults anymore. Whenever a cut set for a hazard H contains the reconfiguration failure, the self-organization mechanism has the potential to delay the occurrence of H . It is especially important to check whether there are any other minimal cut sets for H that do *not* contain the reconfiguration failure; in that case, the system's reconfiguration mechanism can be bypassed by other intolerable faults, which is problematic particularly if there are any singleton cut sets for H . On the other hand, the system is likely to have hazards for which *no* minimal cut sets contain the reconfiguration failure, showing that the self-organization mechanism was not designed to cope with these hazards. The minimal cut sets therefore show the limits of the system's self-organization mechanism and give hints as to where the system's design could be improved. For instance, additional

fault discovery mechanisms could broaden the reach of the self-organization mechanism, effectively turning some of the intolerable faults into tolerable ones.

There are two safety hazards that are most relevant for the case study: The first hazard occurs when a workpiece is damaged, that is, a workpiece is processed in an incorrect order. Its minimal cut sets are missed reconfiguration updates for each robot and cart as well as carts that move to the wrong robots. The reconfiguration failure, on the other hand, does not occur in any of the minimal cut sets, showing that the system is not designed to reduce the first hazard's likelihood. The second hazard occurs when some workpiece is never finished, i.e., not processed any further. The reconfiguration failure is one of the hazard's minimal cut sets, as no further processing is done on any workpiece once a reconfiguration failed. Additional minimal cut sets of the second hazard are malfunctioning sensors that no longer detect the arrival of a workpiece, stalling any further processing. All of the minimal cut sets of both hazards are singletons, showing the weak points of the self-organizing production cell from a safety point of view.

VI. RELATED WORK

There are various academic and commercial tools available for modeling and analyzing safety-critical systems such as MODELICA, SCADE, the COMPASS TOOLKIT, HIP-HOPS, the SAML tool, and many others [17]–[21]. Generally, all of these tools could be used with the approach presented in this paper; compared to S#, however, their lack of a built-in runtime model adaptation mechanism would have to be compensated with additional tools. In particular, Gudemann performed a DCCA of our case study modeled with SAML [17]. A full version of the system such as the one depicted in Fig. 2 was used to determine the limits of the self-organization mechanism, increasing the model checking time needlessly by considering irrelevant state information of the robots, carts, and workpieces. The convoluted specification of the hazard (the hazard has to occur multiple times before it is actually considered to be a hazard) necessitated a more complex DCCA variant which our more systematic modeling and analysis approach does not require.

Runtime models have already been used to analyze adaptive systems for functional correctness and various quality attributes. However, a systematic approach for runtime safety analyses still seems to be an open research question [2]. S# contributes a new idea in this area through its unconventional decision to use the .NET runtime and type system as its meta-metamodel. The choice seems justified, however, as .NET was specifically designed for efficient program execution and runtime object composition, both of which form the basis of S#'s support for runtime model adaptation. But as S# was initially not conceived for runtime safety analyses, it would certainly benefit from incorporating more ideas from the runtime modeling and analysis communities. More research in the area of runtime modeling has been done on runtime safety certification [6], [22], which we currently do not focus on. A completely different approach takes ideas from control theory to dynamically reduce the failure probabilities of adaptive systems [23]. We are not sure if this approach can be applied to self-organizing systems, considering that their possible configurations typically cannot be enumerated in advance.

The work by Gerasimou et al. [24] on increasing the efficiency of runtime analyses appears to be particularly well suited for integration with our approach: A caching strategy could be used to recompute cut sets more efficiently after the occurrences of faults by reusing previously computed cut sets as likely candidates for the new ones. A lookahead strategy could precompute the cut sets for the most likely successor configurations before the next reconfiguration has even started. Statistical model checking would only try to compute the most problematic minimal cut sets instead of conducting a complete analysis, which might be an acceptable tradeoff in order to noticeably increase analysis efficiency. Another approach suggests to split runtime analyses of adaptive systems into two steps [25]: Design time only model checking is used to precompute parameterized expressions that can subsequently be evaluated more efficiently at runtime when the concrete parameter values are known. The expressions can also be used for sensitivity analyses, allowing the system to reason about the impact that changes to the parameters have on the overall system. It is an open question, however, whether large changes to the structure of a self-organizing system require runtime recomputations of the parameterized expressions, which would make the approach less effective.

VII. CONCLUSION AND FUTURE WORK

The need for integrated tools that allow for runtime safety analyses of self-organizing systems is well recognized [3]–[6]. This position paper presents first steps towards a solution based on S#’s support for runtime model adaptation. For the future, we plan to systematically evaluate the analysis speedups of our approach and its applicability to other self-organizing systems. Additionally, we want to investigate the possible benefits of runtime safety analyses for the *Restore Invariant Approach* [26]: The minimal cut sets characterize the *corridor of correct behavior* within which a self-organizing system can reconfigure itself; occurrences of minimal cut sets represent an irrevocable violation of that corridor. We intent to generate *soft constraints* [27] from the minimal cut sets that help the system to reconfigure more “smartly”, possibly increasing the time to the eventual violation of the corridor. Conversely, when testing self-organizing systems [28], the knowledge of the cut sets can be used to generate boundary test cases that force the system to reconfigure in more difficult situations, potentially exposing more implementation bugs in the form of unjustified and unexpected violations of the corridor.

REFERENCES

- [1] A. Habermaier, M. Gudemann, F. Ortmeier, W. Reif, and G. Schellhorn, “The ForMoSA Approach to Qualitative and Quantitative Model-Based Safety Analysis,” in *Railway Safety, Reliability, and Security*. IGI Global, 2012, pp. 65–114.
- [2] A. Bennaceur et al., “Mechanisms for Leveraging Models at Runtime in Self-adaptive Software,” in *Models@run.time*. Springer, 2014, pp. 19–46.
- [3] D. Weyns, M. Iftikhar, D. de la Iglesia, and T. Ahmad, “A Survey of Formal Methods in Self-Adaptive Systems,” in *Proc. of C3S2E*. ACM, 2012, pp. 67–79.
- [4] D. Weyns, “Towards an Integrated Approach for Validating Qualities of Self-adaptive Systems,” in *Proc. of WODA*. ACM, 2012, pp. 24–29.
- [5] R. de Lemos et al., “Software Engineering for Self-Adaptive Systems: A Second Research Roadmap,” in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.
- [6] M. Trapp and D. Schneider, “Safety Assurance of Open Adaptive Systems – A Survey,” in *Models@run.time*. Springer, 2014, pp. 279–318.
- [7] A. Habermaier, J. Leupolz, and W. Reif, “Executable Specifications of Safety-Critical Systems with S#,” in *Proc. of DCDS*. IFAC, 2015, pp. 60–65.
- [8] T. Vogel and H. Giese, “On Unifying Development Models and Runtime Models,” in *Proc. of MoDELS*. CEUR-WS.org, 2014, pp. 5–10.
- [9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” *Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.
- [10] 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*. Springer, 2013, pp. 80–111.
- [11] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck, “Towards a generic observer/controller architecture for Organic Computing,” *GI Jahrestagung*, vol. 93, pp. 112–119, 2006.
- [12] G. Anders, H. Seebach, F. Nafz, J.-P. Steghöfer, and W. Reif, “Decentralized Reconfiguration for Self-Organizing Resource-Flow Systems Based on Local Knowledge,” in *Proc. of EASE*. IEEE, 2011, pp. 20–31.
- [13] M. Becker, S. Kemmann, and K. C. Shashidhar, “Integrating Software Safety and Product Line Engineering using Formal Methods: Challenges and Opportunities,” in *Proc. of SPLC*, vol. 2, 2010, pp. 129–136.
- [14] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.
- [15] W. Vesely, J. Dugan, J. Fragola, Minarick, and J. Railsback, “Fault Tree Handbook with Aerospace Applications,” NASA, Tech. Rep., 2002.
- [16] J. Zhang and B. Cheng, “Model-based Development of Dynamically Adaptive Software,” in *Proc. of ICSE*. ACM, 2006, pp. 371–380.
- [17] M. Gudemann, *Qualitative and Quantitative Formal Model-Based Safety Analysis*. Magdeburg, Univ., 2011.
- [18] Modelica Association, *Modelica – A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.3*, 2014.
- [19] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund, “Designing Safe, Reliable Systems Using Scade,” in *Leveraging Applications of Formal Methods*. Springer, 2006, pp. 115–129.
- [20] 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*. Springer, 2009, pp. 173–186.
- [21] S. Sharvia and Y. Papadopoulos, “Integrating model checking with HiP-HOPS in model-based safety analysis,” *Reliability Engineering & System Safety*, vol. 135, pp. 64–80, 2015.
- [22] D. Schneider and M. Trapp, “Conditional Safety Certification of Open Adaptive Systems,” *ACM TAAS*, vol. 8, no. 2, pp. 1–20, 2013.
- [23] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio, “Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements,” in *Proc. of ASE*, 2011, pp. 283–292.
- [24] S. Gerasimou, R. Calinescu, and A. Banks, “Efficient Runtime Quantitative Verification Using Caching, Lookahead, and Nearly-optimal Reconfiguration,” in *Proc. of SEAMS*. ACM, 2014, pp. 115–124.
- [25] A. Filieri, G. Tamburrelli, and C. Ghezzi, “Supporting Self-adaptation via Quantitative Verification and Sensitivity Analysis at Run Time,” *IEEE Transactions on Software Engineering*, no. 99, 2015.
- [26] M. Gudemann, F. Nafz, F. Ortmeier, H. Seebach, and W. Reif, “A Specification and Construction Paradigm for Organic Computing Systems,” in *Proc. of SASO*, 2008, pp. 233–242.
- [27] A. Schiendorfer, J.-P. Steghöfer, A. Knapp, F. Nafz, and W. Reif, “Constraint Relationships for Soft Constraints,” in *Research and Development in Intelligent Systems XXX*. Springer, 2013, pp. 241–255.
- [28] B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif, “Towards Testing Self-organizing, Adaptive Systems,” in *Testing Software and Systems*. Springer, 2014, pp. 180–185.