Evolving Distributed Embedded Applications during Operation

Kilian Telschig

Dissertation

Faculty of Applied Computer Science
University of Augsburg

2023

Supervisor:	Prof. Dr. Alexander Knapp
Co-supervisor:	Prof. Dr. Bernhard Bauer
Date of submission:	August 10, 2022
Date of disputation:	December 1, 2022

Abstract

Evolving Distributed Embedded Applications during Operation

by Kilian Telschig

The availability of third-party apps is among the key success factors for software ecosystems: The users benefit from more features and innovation speed, while third-party solution vendors can leverage the platform to create successful offerings. However, this requires a certain decoupling of engineering activities of the different parties not achieved for distributed control systems, yet. While late and dynamic integration of third-party components would be required, resulting control systems must provide high reliability regarding real-time requirements, which leads to integration complexity. Closing this gap would particularly contribute to the vision of software-defined manufacturing, where an ecosystem of modern IT-based control system components could lead to faster innovations due to their higher abstraction and availability of various frameworks. Therefore, this thesis addresses the research question: How we can use modern IT technologies and enable independent evolution and easy third-party integration of software components in distributed control systems, where deterministic end-to-end reactivity is required, and especially, how can we apply distributed changes to such systems consistently and reactively during operation?

This thesis describes the challenges and related approaches in detail and points out that existing approaches do not fully address our research question. To tackle this gap, a formal specification of a runtime platform concept is presented in conjunction with a model-based engineering approach. The engineering approach decouples the engineering steps of component definition, integration, and deployment. The runtime platform supports this approach by isolating the components, while still offering predictable end-to-end real-time behavior. Independent evolution of software components is supported through a concept for synchronous reconfiguration during full operation, i.e., dynamic orchestration of components. Time-critical state transfer is supported, too, and can lead to bounded quality degradation, at most. The reconfiguration planning is supported by analysis concepts, including simulation of a formally specified system and reconfiguration, and analyzing potential quality degradation with the evolving dataflow graph (EDFG) method. A platform-specific realization of the concepts, the real-time container architecture, is described as a reference implementation. The model and the prototype are evaluated regarding their feasibility and applicability of the concepts by two case studies. The first case study is a minimalistic distributed control system used in different setups with different component variants and reconfiguration plans to compare the model and the prototype and to gather runtime statistics. The second case study is a smart factory showcase system with more challenging application components and interface technologies. The conclusion is that the concepts are feasible and applicable, even though the concepts and the prototype still need to be worked on in future – for example, to reach shorter cycle times.

Kurzfassung (German)

Anpassen verteilter eingebetteter Anwendungen im laufenden Betrieb

von Kilian Telschig

Eine große Auswahl von Drittanbieter-Lösungen ist einer der Schlüsselfaktoren für Software Ecosystems: Nutzer profitieren vom breiten Angebot und schnellen Innovationen, während Drittanbieter über die Plattform erfolgreiche Lösungen anbieten können. Das jedoch setzt eine gewisse Entkopplung von Entwicklungsschritten der Beteiligten voraus, welche für verteilte Steuerungssysteme noch nicht erreicht wurde. Während Drittanbieter-Komponenten möglichst spät – sogar Laufzeit – integriert werden müssten, müssen Steuerungssysteme jedoch eine hohe Zuverlässigkeit gegenüber Echtzeitanforderungen aufweisen, was zu Integrationskomplexität führt. Dies zu lösen würde insbesondere zur Vision von Software-definierter Produktion beitragen, da ein Ecosystem für moderne IT-basierte Steuerungskomponenten wegen deren höherem Abstraktionsgrad und der Vielzahl verfügbarer Frameworks zu schnellerer Innovation führen würde. Daher behandelt diese Dissertation folgende Forschungsfrage: Wie können wir moderne IT-Technologien verwenden und unabhängige Entwicklung und einfache Integration von Software-Komponenten in verteilten Steuerungssystemen ermöglichen, wo Ende-zu-Ende-Echtzeitverhalten gefordert ist, und wie können wir insbesondere verteilte Änderungen an solchen Systemen konsistent und im Vollbetrieb vornehmen?

Diese Dissertation beschreibt Herausforderungen und verwandte Ansätze im Detail und zeigt auf, dass existierende Ansätze diese Frage nicht vollständig behandeln. Um diese Lücke zu schließen, beschreiben wir eine formale Spezifikation einer Laufzeit-Plattform und einen zugehörigen Modell-basierten Engineering-Ansatz. Dieser Ansatz entkoppelt die Design-Schritte der Entwicklung, Integration und des Deployments von Komponenten. Die Laufzeit-Plattform unterstützt den Ansatz durch Isolation von Komponenten und zugleich Zeit-deterministischem Ende-zu-Ende-Verhalten. Unabhängige Entwicklung und Integration werden durch Konzepte für synchrone Rekonfiguration im Vollbetrieb unterstützt, also durch dynamische Orchestrierung. Dies beinhaltet auch Zeit-kritische Zustands-Transfers mit höchstens begrenzter Qualitätsminderung, wenn überhaupt. Rekonfigurationsplanung wird durch Analysekonzepte unterstützt, einschließlich der Simulation formal spezifizierter Systeme und Rekonfigurationen und der Analyse der etwaigen Qualitätsminderung mit dem Evolving Dataflow Graph (EDFG). Die Real-Time Container Architecture wird als Referenzimplementierung und Evaluationsplattform beschrieben. Zwei Fallstudien untersuchen Machbarkeit und Nützlichkeit der Konzepte. Die erste verwendet verschiedene Varianten und Rekonfigurationen eines minimalistischen verteilten Steuerungssystems, um Modell und Prototyp zu vergleichen sowie Laufzeitstatistiken zu erheben. Die zweite Fallstudie ist ein Smart-Factory-Demonstrator, welcher herausforderndere Applikationskomponenten und Schnittstellentechnologien verwendet. Die Konzepte sind den Studien nach machbar und nützlich, auch wenn sowohl die Konzepte als auch der Prototyp noch weitere Arbeit benötigen – zum Beispiel, um kürzere Zyklen zu erreichen.

Acknowledgements

I would like to thank my supervisor Prof. Alexander Knapp. Thank you for your support of this work – your guidance, your perseverence, and your faith in the topic. I would also like to thank my co-supervisor Prof. Bernhard Bauer, my co-disputant Prof. Jörg Hähner, and all people from the University of Augsburg who supported and challenged my work. Thank you to Prof. Wolfgang Reif and everyone from the Institute for Software & Systems Engineering for the chair retreats ("Hütte") with all the interesting presentations, discussions and sociable evenings. Some of your research is referred in this thesis and I will definitely keep an eye on it. I would like to thank my supervisors, mentors, and colleagues at Siemens for helping me grow professionally and personally. Thanks Dietmar, for getting me hooked on architecture research in the automation domain by your passion and foresight. Thanks Konstantin, for sharing your advice and experience, and for our philosophical discussions. Thanks Sebastian and Andreas S., for taking on the challenge of supervising me and for stressing the bigger picture and the ecosystem perspectives of my research. And thanks Bene for your advice, especially when I started the industrial PhD. Thanks Nicole and Andreas B. for supporting me and my topic both before and after I started as a full-time employee at Siemens. I would also like to thank Fabian and all the other colleagues from the PhD circle for all the formats in which we exchanged research ideas and concepts across fields. Finally, I have to thank my family and friends for their understanding the many times I had to focus on my work. I would like to thank my "doctor brother" André in particular and also Philip for the invaluable tuesdaily and all the on- and off-topic discussions. I also thank my sister Carmen for our quality time that helped to stay on track in tough times. And thank you to Claudia, who lovingly supported me through all the years. Without your support this thesis would likely not exist!

Contents

Chapter 1

Introduction

Software-defined manufacturing is predicted to reshape the manufacturing landscape of the future [XCG+21]. According to this, several national and international initiatives such as Industry 4.0 [SS16] are currently achieving a separation between physical manufacturing layers and their usage by (cloud) connected software. The extrapolated next step is resource sharing and collaboration across manufacturing sectors to ultimately schedule production workloads freely on a unified global manufacturing platform. Production as a service [BNO+21] is fully unleashed when the derivation of concrete production processes and allocation to suitable smart factories is solved automatically on-demand (see figure 1.1). One precondition for this vision is the availability of a large number of "flexible and universal manufacturing nodes [...] that can produce a wide range of products and easily switch from one product to another" [XCG+21]. Thus, there is a lot of research on assumed technical enablers for this, for instance artificial intelligence (AI), information and communications technology (ICT), (industrial) internet of things (IoT/IIoT), blockchain, but also, robotics and 3D printing. However, we attack a less frequently considered technical gap in the runtime platform close to and even immediately controlling the technical process. But before we get into that, we briefly recap the current runtime approaches in factory automation (we will go more into detail and include other domains such as avionic and automotive in chapter 2).

At its core a factory is an industrial control system that uses primary equipment to solve a production problem. The kind and timing of the equipment's engagement – especially the coordination of the equipment – is often still solved by mechanics an the like, but of course also by industrial control software. According to the traditional automation pyramid [ISO13], the field-level control software on Programmable Logic Controllers (PLCs) cyclically calculates outputs from inputs. It uses I/O modules to communicate with the primary equipment and with other PLCs, e.g. to provide commands and setpoints or to directly actuate physical components. This process can be monitored and manipulated from Supervisory Control and Data Acquisition (SCADA) systems and especially Human-Machine-Interfaces (HMIs). Higher-level control (e.g. decision on production quantities) may be possible in pre-defined ways from corresponding IT systems, i.e., Manufacturing Execution System (MES) and Enterprise Resource Planning (ERP). This is a common system architecture, especially for high-throughput repetitive automation systems with low required flexibility and adaptability in the use of the primary equipment. Most often, it is important for the purpose of such a system that the end-to-end reactivity from sensors over controllers to actuators is bounded (sometimes this is even safety-critical). Figure 1.2 shows a small sorting system that could be used in a bottling plant. A barcode scanner detects when a specific product unit is passing the conveyor – in this case a bottle. Shortly behind the location a switch should be able

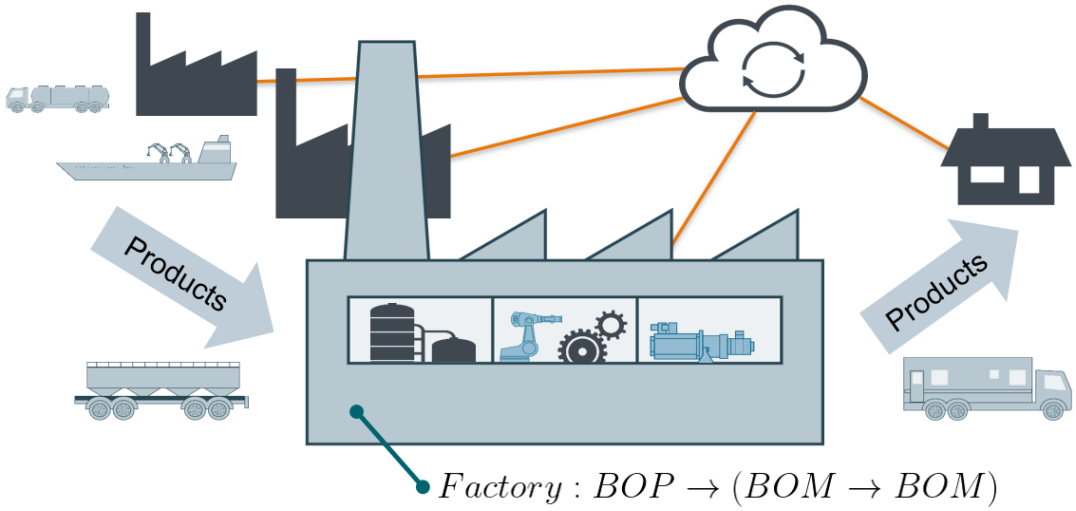

FIGURE 1.1: Production as a service: Automatic value chains across smart factories could solve production problems on-demand with low specific engineering effort by reasoning about possibilities to produce a product based on its bill-of-process (BOP) and bill-of-materials (BOM), or even directly on its digital twin.

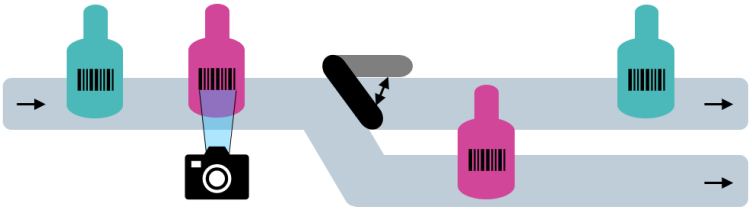

FIGURE 1.2: A (not so) plain sorting system in a bottling plant: Redirect selected bottles to another conveyor using a switch and a barcode scanner. The routing decision is time-critical, but additional information from other systems is needed.

to redirect the bottle to another conveyor if needed, e.g., if it contains a certain beverage. The success depends on the conveyor speed, the distance between bottles, and the effective reactivity of the switch. Thus, the time for the calculation of the routing decision and delivery of the command may be in the range of a few milliseconds. However, the bottle's destination must be determined from the barcode, e.g. via an OPC UA [ISO20] request to the SCADA system or a direct database query. This example shows a typical automation problem for traditional distributed control systems, but also, how IT systems are already used in time-critical decisions.

In more recent approaches towards more flexible and adaptable production systems, the lower layers of the pyramid may collapse to more "autonomous" cyber-physical production systems (CPPS), special cyber-physical systems (CPS) to enable Industry 4.0 [Mon14]. Figure 1.3 shows a sketch of such a system, a variant of which is studied more in detail in section 6.2. In a keynote at WICSA and CompArch 2016, Ghezzi abstracted the challenge of architecting safe, secure and reliable CPSs (cf. [TKG16]) as fulfilling the following formula (based on Zave and Jackson [ZJ97], but with environment E, software S, and requirements R):

$$E \& S \models R$$

In terms of this model, we distinguish CPPSs from traditional automation in that

they better work in less limited environments E due to their more self-reflective software S, which can reason about how to fulfill (potentially dynamic) requirements R. CPPS are usually controlled by a MAPE-K loop program to achieve more autonomy in terms of the self-* abilities (self-optimization, self-healing, ...) [KC03]: The CPPS continuously monitors the sensor input, analyzes the current situation, plans required actions (e.g. based on goals and rules) and then adapts the execution of its actions – which may all involve internal knowledge. These systems are more autonomous in the sense that they can operate in a wider decision space only limited by the capabilities of their equipment and their reasoning. The promising benefit of autonomy is that less engineering may be required to make the CPPS contribute to the production processes, and that these could be adapted faster. However, besides using different internal architectures and technologies, CPPSs may also require a higher level of (semantic) communication and interaction with other CPPSs. Such multi-CPS-systems can be found in literature as multi-agent systems (MAS) [DKJ18], collaborative embedded systems (CrESt) [BBK+21], or collaborative intelligent systems (CIS) [BNO+21], to mention just a few. To avoid that the autonomy is limited by special-purpose interfaces between CPPSs, common languages (or: ontologies) are introduced, so the CPPS understand each other [Woo09]. Due to all these aspects, modern IT technologies are becoming more important in control loops, because they provide rich libraries and abstractions that help to solve the complexity of CPPS. However, IT technology is in general less deterministic, which leads to reliability problems in time-critical control systems. Moreover, as already stated by Zoitl [ZLMV10], much work towards flexible and adaptable systems was focusing on the higher-level control (planning and scheduling of the production process), whereas the flexibility and adaptability of the lower-level real-time control executing the process is usually restricted. Even with the newer IT-based approaches, the adaptability on lower levels during production is limited to choosing from available skills (corresponding to ontologically modeled actions taken in the MAPE-K loop above, cf. [MBW+18]) and often only in idle production phases when no hard deadlines exist. This in turn limits the higher-level adaptability – e.g. the logic for reasoning about skills can not be adapted during operation. This is not acceptable considering the long lifetime and high cost of downtime of factories versus obsolescence of IT technologies, especially regarding security when they are more and more cloud connected. Additionally, not being able to update all the application parts frequently simply does not leverage the full innovation speed of modern IT technologies. Thus, reconfiguration of distributed embedded applications as a whole – regardless of their concrete architecture (e.g. agents, skills etc.) – should be possible during operation without or at least with deterministic degradation of the system's end-to-end reactivity. This problem was first tackled by our work and still there is little research into this direction.

This thesis describes our concepts to get closer to "classical" determinism with "modern" technologies to improve the engineering efficiency, abstraction of control, and lower-level adaptability of distributed embedded applications. In particular, we propose a component-based runtime platform concept and reconfiguration mechanism that is independent from concrete patterns in the architecture of the distributed embedded application and the organization of the equipment in terms of collaboration. We also want to make it possible to solve collaborative production problems by distributed embedded applications that span across CPPSs (even from different vendors), but also including traditional automation system parts. This would enable distributed solutions even with less powerful semantic languages and models, as the interfaces would not need to cover all the production problems that might

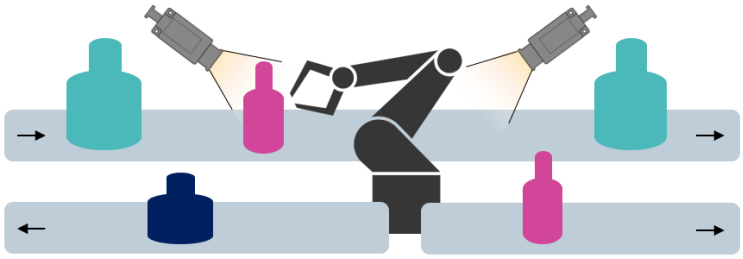

FIGURE 1.3: A CPPS in a bottling plant: A robot uses cameras to detect bottles of different colors and shapes and redirects them based on reasoning about where and how to place it, possibly even including "discussions" with other systems.

occur in the various domains in such a virtually integrated distributed embedded application. Therefore, software components and distributed embedded applications should be reusable in different constellations of CPPS due to abstraction (from concrete equipment, from concrete network topologies, ...). To make this possible, CBSE engineering activities are decoupled, e.g. distributed embedded applications can be defined independently from a concrete deployment and are allocated to computing nodes in a later engineering step – a concept known from Automotive Open System Architecture (AUTOSAR) [AUT15]. We think that this is a necessary step as a basis for an ecosystem of independent hardware and software suppliers for rapidly adaptable higher-level distributed control systems. Obviously, there are many challenges arising from this, that we have to exclude from the scope of this thesis. The next sections will describe the concrete scope and contribution of this thesis, a running example, and the outline.

1.1 Research Scope and Contribution

The subject of this thesis is how we can use modern IT technologies and enable independent evolution and easy third-party integration of software components in distributed control systems, where deterministic end-to-end reactivity is required, and especially how we can apply distributed changes to such systems consistently and reactively during operation. We explain the different aspects of this subject before we list our concrete contribution. Modern IT technologies such as neural networks can contribute to achieving higher-level control due to their abstraction and reuse potential, faster innovation cycles, and an increasing number of talents. However, they are not designed for use in time-deterministic systems. We are looking for approaches that are widely independent from programming platforms and communication technologies used by the software components (so that any existing or upcoming framework can be leveraged transparently by applications), and which make them useable in real-time control loops. It is important for the speed of system development that consistent integration and reuse of software components is very easy, despite hard end-to-end real-time requirements. We expect system adaptations to happen more frequently due to shorter lifecycles of (reused) components and the nature of upcoming classes of control systems, if we can avoid the high cost of downtime during reconfigurations. Thus, the problem arises what design- and runtime concepts can be used to achieve system adaptations during full operation. This is required for the adaptation of non-stop control systems, too, e.g. when a shutdown would lead to deformation of the equipment. However, we don't only want to replace individual components that are backwards-compatible, but even

change multiple components simultaneously if required or even restructure a distributed embedded application. In the extreme case, it should even be possible to replace a distributed embedded application with a completely different one during operation. Therefore, we must be able to change the set of component instances, their communication relationships, their I/O access, and support state transfer. A fundamental architectural problem resulting from these integration and adaptation goals is how the software components can stay as independent as possible over all phases from development over integration until execution while the resulting system must fulfill real-time requirements. However, we also describe some limitations of the scope of this thesis. First, we exclude self-reconfiguration, safety, security, and supporting aspects such as billing and intellectual property protection, even though we tried to make our concepts extensible for those aspects in later iterations. As we target modern IT technologies, which have less predictable execution behavior, the scope of real-time requirements is limited to the millisecond-scale regarding deadlines (not microseconds). While function blocks are widely used in control software today and could be related to software components, they are too fine grained for a runtime platform for modern technologies. A software component based on such classical technology would more likely be a function block diagram. In terms of object-oriented programs, the granularity of a software component is not an object or a set of objects, but a (potentially multi-threaded) program. Finally, our aim is not to define the most feature-rich system and component model or support specific widely used middleware technologies. Instead, we keep the system and component model minimalistic and rather achieve a break-through in the dynamic integration over all phases. That means when it comes to the features required from the runtime platform to achieve such a break-through, we are interested in understanding and modeling in detail how distributed real-time integration and dynamic reconfiguration of a modular application can be enabled.

Contribution. We make following contribution to this area of research:

- An overview and discussion of important concepts from the interrelated – but to some degree disjoint – fields of software architecture, real-time systems, and formal methods, including existing architectures for distributed embedded systems and concepts for dynamic reconfiguration.
- A conceptual framework based on Real-Time Maude for formal specification and analysis of modular distributed real-time systems and their dynamic reconfiguration consisting of a design-time model with a component model, a distributed embedded application model, a hardware model, a network topology model, a deployment model, and a reconfiguration plan model, and a runtime model with a model of application containers, of network communication, of a real-time container agent, and of reconfiguration plan execution.
- An evolving dataflow graph as part of the conceptual framework, which is an abstract transition system of the behavior of a distributed embedded system including ongoing reconfigurations that can be constructed from a given instance of the component model.
- Reconfiguration blueprints for systematically deriving reconfiguration plans for specific reconfiguration use cases.

- An architecture documentation of the real-time container architecture, which complies to the formal runtime platform model and supports modern IT technologies within software components.
- An evaluation of the conceptual framework and the real-time container architecture prototype via two case studies, which analyze and demonstrate how the formal analysis can be used for systems based on the platform-specific realization, how the real-time container architecture can be extended for specific industrial I/O technologies, how modern IT technologies are supported within containers, how well engineering steps can be decoupled by the conceptual framework, and what runtime behavior can be achieved based on a 24 hour benchmark run with a reconfiguration loop.

Publications. Throughout the work on this contribution, we published following papers that already describe some of the concepts in earlier phases and less formally:

ETFA 2017: Kilian Telschig and Alexander Knapp. *Towards Safe Dynamic Updates of Distributed Embedded Applications in Factory Automation.* [TK17]

Why safe updates of distributed embedded systems without downtime will be required in factory automation and how it could be achieved with time determinism and modern technologies by means of real-time containers (vision).

CASE 2018: Kilian Telschig, Andreas Schönberger, and Alexander Knapp. *A Real-Time Container Architecture for Dependable Distributed Embedded Applications.* [TSK18]

Informal description of our concrete real-time container architecture (engineering, architecture, runtime behavior) for easy integration of distributed control systems from software components (without dynamic reconfiguration) and first benchmark measurements.

ICSA 2019: Kilian Telschig and Alexander Knapp. *Synchronous Reconfiguration of Distributed Embedded Applications During Operation.* [TK19a]

Reconfiguration extensions for the real-time container architecture, especially how exactly dynamic updates are specified and executed synchronously across nodes based on reconfiguration plans (without state transfer).

INDIN 2019: Kilian Telschig and Alexander Knapp. *Time-Critical State Transfer during Operation of Distributed Embedded Applications.* [TK19b]

State transfer extensions for the real-time container architecture, i.e., additional reconfiguration primitives/steps and examples how to use them to maintain the state of the software components either without downtime or with deterministic quality degradation.

INDIN 2022: Joseph Hirsch, Marius Lichtblau, Marian Lingsch Rosenfeld, Kilian Telschig, and Alexander Knapp. *Cube Bot – A Smart Factory Showcase for the Real-Time Container Architecture.* [HLL+22]

A proof-of-concept showing that the I/O handling and component model of the real-time container architecture is suitable for technologies in upcoming smart factory systems, i.e., a straight-forward way to add support for using USB webcams and an RS-232 interface of a robot by containerized components

by I/O extensions according to the philosophy of the real-time container architecture, and an application that uses a neural network and the API of the robot to realize a *Cube Bot* system which sorts cubes by colors.

In close proximity of the contribution, but beyond the exact thesis scope we published following additional papers:

WICSA 2016: Kilian Telschig, Nikolai Schöffel, Klaus-Benedikt Schultis, Christoph Elsner, and Alexander Knapp. *SECO Patterns: Architectural Decision Support in Software Ecosystems.* [TSS+16]

A pattern language for software ecosystem patterns and a methodology to systematically derive architectural decisions based on ecosystem patterns, an ecosystem classification, and an architecture knowledge base. In this thesis we contribute platform concepts to technically enable ecosystem scenarios in industrial domains, too.

Computer 7/2021: Arne Bröring, Christoph Niedermeier, Ioana Olaru, Ulrich Schöpp, Kilian Telschig, and Michael Villnow. *Toward Embodied Intelligence: Smart Things on the Rise.* [BNO+21]

A foresight study of likely industrial application scenarios for artificial embodied intelligence in the next two decades, including how this would disrupt these domains and influence megatrends. In this thesis we contribute platform concepts that could be used for the technical realization of some of the proposed scenarios, e.g. production as a service.

EuroPLoP 2022: Joachim Fröhlich, Steffen Klepke, Christoph Stückjürgen, and Kilian Telschig. *Seamless Upgrade: Upgrade Functions Executing on a Control System.* [FKST23]

A pattern to enable harmless upgrades of individual software functions using in-field testing and a coordinated switch-over. In this thesis we also enable upgrades of individual functions, but this approach also enables compatibility-breaking updates of multiple functions in the distributed control system. In turn, we do not (yet) allow the application to influence the reconfiguration timing to consider application-specific conditions of the technical process.

1.2 Running Example: The *onBtnSwitch* System

We describe *onBtnSwitch*, a small "hello world" kind of distributed control system first described in [TSK18]. We use it as running example throughout this thesis, e.g. to show some instantiations of the different model fragments and to demonstrate different reconfiguration use cases and analyses. We also use it in parts of the evaluation, especially for comparing the model-based analysis and the prototype as well as for a benchmark reconfiguration loop. The simplicity of *onBtnSwitch* makes it a good running example, because the core subject of this thesis is our runtime platform concept and not application-specific considerations.

Figure 1.4 shows an overview of *onBtnSwitch*. Two nodes (*node1* and *node2*) are connected via Ethernet. We use SIMATIC IOT2040 devices with x86-based CPUs running at 400 MHz and with 1 GB RAM. The microcontrollers have a USER Button and a USER LED, which are accessible via GPIO (in our case *gpio63* and *gpio13*). An embedded application distributed over the two nodes should invert the state

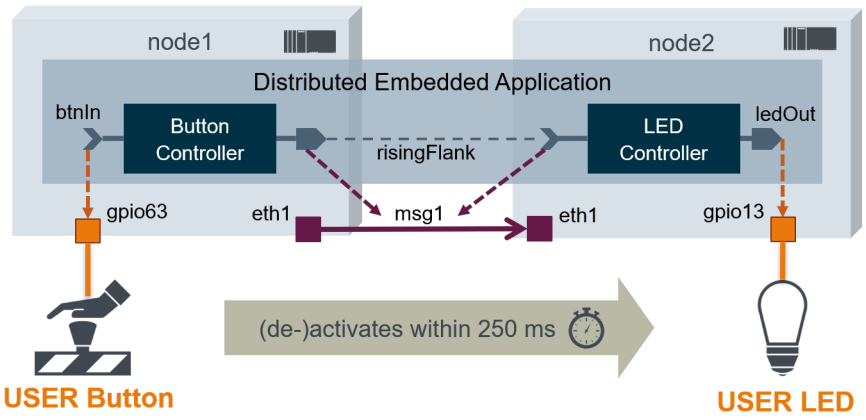

FIGURE 1.4: The running example system *onBtnSwitch*. Two micro-controllers are connected via Ethernet. When the USER button is pressed (rising edge) on the first node, the USER LED on the second node shall be switched on/off within 250 ms. This is done by two components of a distributed embedded application.

of the USER LED on *node2* whenever the USER button on *node1* is pressed (rising edge). We assume that the end-to-end reactivity of this inversion from any button press until the effective LED switch is time-critical and must be done within 250 ms, for instance. An additional engineering system *ES* is used to start and monitor the system and to trigger reconfigurations. It is usually only connected to one of the nodes and reaches the other one via the plant network. As an example reconfiguration, assume we have to simultaneously replace the *Button Controller* component on *node1* and the *LED Controller* component on *node2*, because the interface between them changes from a boolean *risingFlank* to an integer *flankCount*. The GPIOs need to be handed from the old component versions to the new ones during these replacements. Also note that the network message *msg1* becomes larger, which might require simultaneous network reconfiguration, too. This reconfiguration should be done during full operation while deterministically maintaining the 250 ms reactivity, but including state transfer if needed. Section 6.1 describes more reconfiguration scenarios for *onBtnSwitch*.

Despite the simplicity of *onBtnSwitch* it still demonstrates the conflicting requirements that should be addressed by our runtime platform:

Distributed: The application is distributed over two nodes, so there is network communication between at least two software components.

Embedded: The application components need to access the installed equipment (button and LED).

Stateful: The application components might have a state that is needed for computation (e.g. whether the button was already pressed before).

Real-time: The distributed embedded application must provide bounded end-to-end reactivity (250 ms).

Mixed-criticality: Besides the time-critical embedded application, *ES*-related processes for operation features share the same CPU and network (e.g. SSH).

Cross-platform: It should be possible to use the best-fitting platforms and frameworks ("modern IT technology") for each software component independently from the others (e.g. C++ in one component and Python in another one).

Third-party: It should be easy to integrate the different software components, so that third-party components from independent solution vendors could be used (ultimately: reuse components from a public marketplace). In particular, this also leads to the need to isolate components at runtime to limit their potential system impact.

Reconfiguration: Still, we want to update during full operation, even in case of breaking changes that require simultaneous replacements (e.g. the content of the network communication changes from boolean to integer).

Thus, while *onBtnSwitch* might seem over-simplified in the first place, it still shows many challenges that we expect in real distributed control systems. For instance, consider "hand-eye-coordination" of a robot and a camera in a smart factory, where instead of a button input a camera is used and instead of an LED output a robot is controlled. In such more advanced systems, easy integration and bounded end-to-end reactivity is desired, too, but probably more difficult to achieve. In the evaluation in section 6.2 we describe such a system as a more realistic smart factory case study which also uses more challenging technologies.

1.3 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 (Foundations and Related Work) gives an overview of existing approaches to distributed embedded applications and their dynamic reconfiguration.

Chapter 3 (A Component Model for Modular and Evolvable Distributed Embedded Applications) provides a conceptual framework for formal specification and analysis of component-based distributed real-time systems based on Real-Time Maude based on a design-time model and a runtime model of the abstract runtime platform. This chapter focuses on static systems, but already provides some reconfiguration-enabling features such as its timing behavior and reconfiguration hooks.

Chapter 4 (Reconfiguration of Distributed Embedded Applications during Operation) specifies reconfiguration extensions to the static model described in chapter 3, so that such systems can be changed during operation based on reconfiguration plans. This chapter includes methods for analysis of such reconfigurations and reconfiguration blueprints for some kinds of reconfigurations.

Chapter 5 (Evaluation Platform: Real-Time Container Architecture) documents the architecture of a realization of the runtime platform concepts modeled in chapter 3 and chapter 4. It is a reference solution to show that the platform concepts are feasible, especially with regard to the underlying goals and assumptions. A prototype of the real-time container architecture was used for the platform-specific aspects of the subsequent evaluation.

Chapter 6 (Evaluation of the Platform Concepts and Prototype) describes the evaluation of our modeling and runtime platform concepts using our Maude model from chapters 3 and 4 and our platform prototype from chapter 5. The *onBtnSwitch* system was used to gather runtime statistics and to compare the real system with the

simulation. A more challenging *CubeBot* system was used to study the suitability of our engineering and runtime concepts for real distributed control systems.

Chapter 7 (Conclusion) summarizes the results of this thesis and gives an outlook of future work.

Chapter 2

Foundations and Related Approaches

We give an overview of various existing approaches to distributed embedded applications and their dynamic reconfiguration. This intersection is still rare compared to the amount of work in each of the related fields (see figure 2.1). Thus, we cover approaches to distributed control systems without dynamic reconfiguration and related architectures and concepts for dynamic reconfiguration in other kinds of systems. Some of these approaches are the basis for the concepts described in the subsequent chapters. Other approaches are covered to locate our work in the state of the art and to clearly point out the technical gap addressed. We also provide some background on related architectural principles, technologies and formal methods as foundation for this thesis.

We structured the foundations and related work in three blocks. In the first block we elaborate on architecture principles that lead to modularity, maintainability, and evolvability of potentially distributed systems. These principles were taken into account during fundamental design decisions in our concepts, e.g. Component-Based Software Engineering (CBSE). The second block is on runtime platform concepts for industrial control systems. We propose runtime platform concepts for distributed control systems with hard real-time requirements, which support dynamic reconfiguration. Thus, we compare the different architectures and mention existing work to add support for dynamic reconfiguration, where available. Some of the existing concepts are part of the foundation of our work, e.g. the Logical Execution Time (LET) paradigm. We also elaborate on concrete technologies used in our evaluation prototype, such as Embedded Linux and containers. Finally, the third block is on formal methods, which build the basis for our formal specification. This block also covers existing formal component frameworks which support dynamic reconfiguration. As this is a huge field, we only give an overview and focus on those approaches which are at least close to reconfiguration of distributed real-time systems.

2.1 Architecture Principles

"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" [BCK03]. There are some fundamental software architecture principles besides more specific patterns and tactics that lead to "good" architectures in the sense that system goals are achieved. However, new software architecture concepts require that these principles are instantiated, again. This is particularly difficult for distributed control systems, for which we have to define a feasible system architecture, too. "A system's architecture is a representation of a system in which there

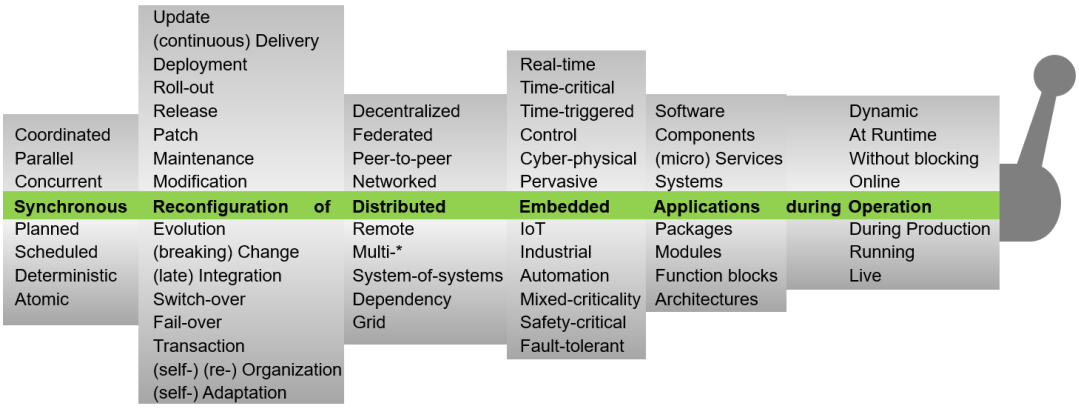

FIGURE 2.1: An overview of related fields by alternative terms for the title of our publication "Synchronous Reconfiguration of Distributed Embedded Applications during Operation" [TK19a]. The terms are not always synonyms, but indicate the broad range of potentially related work. Despite the potential impact, the combination of all these aspects is still rare.

is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and a concern for the human interaction with these components" [BCK03]. Especially in the context of distributed control systems, software architecture principles collide with the non-functional aspects such as real-time requirements. We describe software and system architecture principles, which we consider foundational for a modern approach to distributed control systems.

2.1.1 Model-Driven Engineering (MDE)

A model is an abstraction of a part of reality based on selected artifacts and relationships between them (cf. [FOSS20]). A software and system architecture can also be seen as a model of a system, as it shows the important aspects for understanding a system and its context. Often an architecture in its role as a model is mainly used for reasoning about a system, e.g. based on an underspecified set of agreed terms and ad-hoc drawings, to then implement this by development teams. There may be different views by different stakeholders, i.e., different (hopefully) complementary models or different excerpts of (hopefully) the same model for different purposes. Even the code of a system can be seen as a (computation-centered) model, too, and there should be a strong correlation between the architecture-level model and the code. Model-Driven Engineering (MDE) (see figure 2.2) is an approach to create more formalized, machine-consumable and interlinked models via domain-specific (modeling) languages (DSL/DSML), so that then tool support can be leveraged, e.g. for automated model transformations and even code generation [Sch06]. While there are still many research challenges, MDE is widely used across domains and industries [BCPP20]. The Model Driven Architecture (MDA) defined by the Object Management Group (OMG) provides following principles and specifications so that Model-Driven Engineering (MDE) can be used effectively and interchangeably [OMG14]. A basic principle is that metamodels should be defined for each model, defining what kinds of elements exist and which relationships are allowed. Using the metamodels, general-purpose modeling tools can ensure that a model instantiation is conformant and abstract model transformations and code generators can be defined. In the MDA, MOF [OMG16] and UML [OMG15b] are proposed for

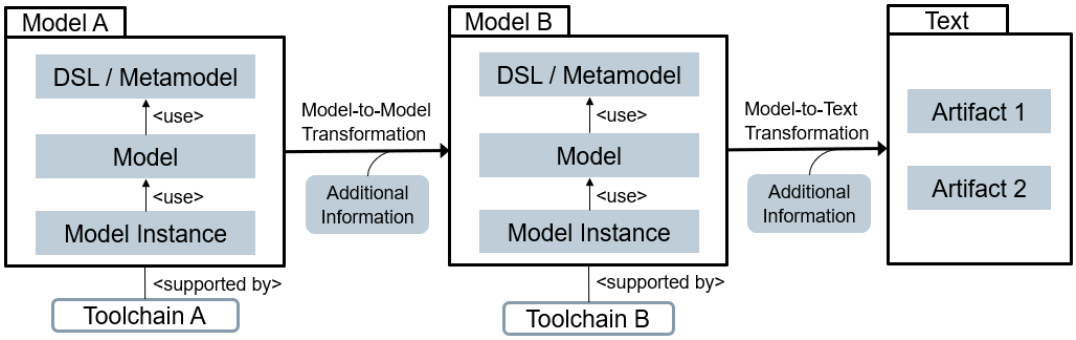

FIGURE 2.2: An abstract overview of MDE based on [GPR06] (fig. 5.1).

metamodeling and modeling independently from specific technologies and tools. In addition to custom DSLs, a wide range of metaprogramming languages and concepts exist, which enable model-driven concepts more integrated with the target programming languages (e.g. reflections or aspects) [LS19]. More abstracted from the code there are Architecture Description Languages (ADL), which focus on architectural entities and the relations between them, e.g. components, interfaces and connectors [MT00]. In general, any set of graphical and text-based DSLs can be combined to provide the most intuitive and efficient modeling experience to the users of the corresponding model fragments, as long as the integration is defined. Besides using well-defined models, the model-based approach should also foster separation of concerns. According to this fundamental principle, any bigger problem should be approached via independently analyzable aspects to make it comprehensible. Dijkstra promoted this principle in 1982 arguing: "[...] even if not perfectly possible, [separation of concerns] is yet the only available technique for effectively ordering one's thoughts that I know of" [Dij82]. It is a prerequisite for sequential and parallel division of labor in software and system engineering. The MDE approach should support separation of concerns by abstraction (transformation patterns, technology independence, ...), layers (e.g. business, logic, implementation) and views/viewpoints (not everything visible in every representation, consistency ensured by model). A major separation should be maintained between a platform-independent model (PIM) and the platform-specific model (PSM). The PSM includes concrete technologies, whereas the PIM is an abstraction that can be translated to different technologies. This has the benefit that technologies can be exchanged with minimized adaptations on the transformation level. The MDA also suggests that models can also distinguish between the representation of the current reality ("as-is") and the desired future reality ("to-be"). In modern approaches towards agility and CI/CD it is even required to include versioning concepts in the MDE approach (cf. [CS21]), e.g. based on semantic versioning [Pre13]. This is particularly important, if we change the system model and want to apply the change to a running instance of the system. However, the models are not only useful with code generators. Consistent and up-to-date documentation can be generated from the model in different delivery forms. And last but not least, the models can and should be used in all stages for analysis of the different system aspects (performance, correctness, ...). If we combine MDE with CBSE (see section 2.1.2), an important aspect of the component model and the runtime platform is to effectively separate this analysis for the different engineering steps.

Based on these concepts, MDE can reduce time, cost and risk while improving the resulting systems' fitness for purpose [OMG14]. The models and automation

can help master the complexity of systems and to become faster, encouraging more frequent changes of systems in an agile manner. Thus, these concepts can be found in our component model and platform concept, and even in the structure of this thesis. Sections 3 and 4 describe our design-time and runtime model in several separately analyzable model fragments along the proposed engineering workflow and in a technology-neutral formal specification. Section 5 describes a platform-specific realization of this specification. The evaluation in section 6.1 describes that a platform-independent formal specification can be used for engineering and analysis of distributed control systems running on the platform-specific platform prototype to some extent. While there is a gap between the formal specification artifacts and the platform-specific artifacts, especially the reconfiguration plan description language shows how close the two sides of the concepts are, so that model transformations and code generation could be achieved in future. In context of our approaches for dynamic reconfiguration, we are still looking for ways to generate a reconfiguration plan to transition to the new system based on a "diff" between two system versions. Reconfiguration planning is a manual step as of today, but the platform-independent reconfiguration blueprints described in section 4.3 are a first step towards more generically solving dynamic reconfiguration problems for distributed control systems in order to automate this complex engineering step.

2.1.2 Component-Based Software Engineering (CBSE)

We recall the separation of concerns principle, to break down problems into aspects. This principle has been applied in many platforms and architecture concepts, especially by different ways to modularize a system. One common modularization concept is CBSE. Following abstract definition of the term *software component* was formulated by Szyperski: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [Szy98]. However, Szyperski also stated that software components and corresponding runtime platforms were not highly standardized across industries, as in other engineering disciplines, and that this is probably due to the immaturity of software engineering compared to other much older disciplines [Szy98]. Standards are still emerging today, even though there are several established domain-specific standards, some of which we cover in section 2.2. Figure 2.3 shows some of the widely used elements of component-based systems. A component runtime platform starts and stops components providing an execution context in which behavioral elements can be executed. The interaction between the component and its environment is only possible via the communication means provided and required by its ports. Different communication and execution concepts by the runtime platforms lead to dedicated more or less formalized component models. If the component model is formalized and covers all required information from design-time to runtime, it can be a good basis for MDE. Thus, the MDE-related principles from section 2.1.1 also apply to component models.

On top of a component model, CBSE defines decoupled structured procedures leveraging modularization and composition techniques built on the strict interfaces of software components and the abilities of the corresponding engineering and runtime platform. According to Janisch, "CBSE is characterized [by] (1) independent development of reusable components (2) system development by assembly and hierarchical composition (3) maintenance and evolution by substitution (and adaptation) of components" [Jan10] (see figure 2.4). The degree to which these engineering

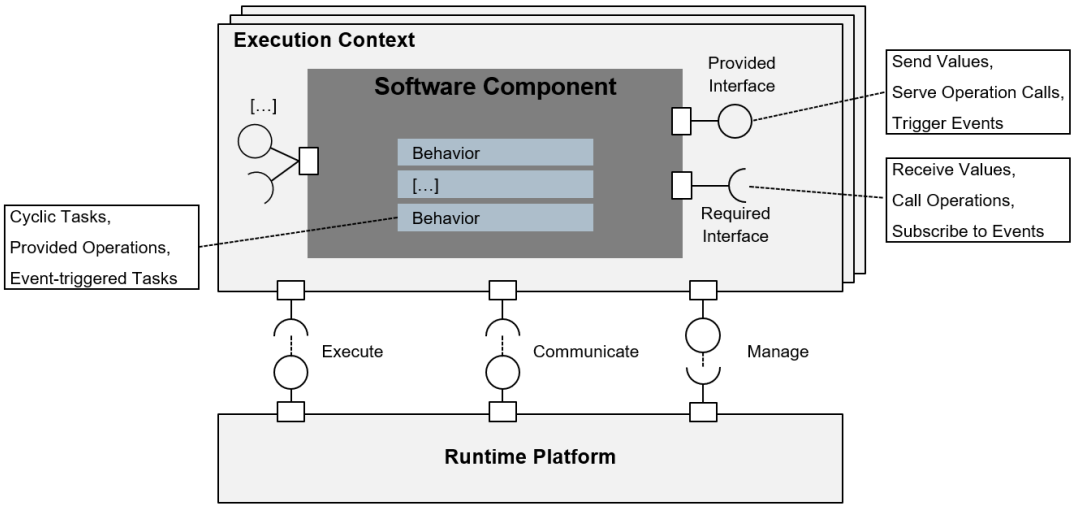

FIGURE 2.3: An abstract view on component-based systems based on [TK17].

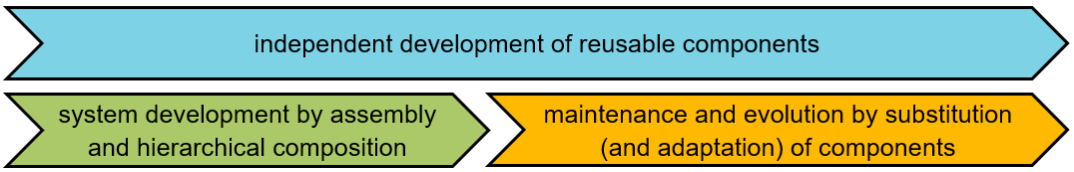

FIGURE 2.4: An abstract view on CBSE based on [Jan10].

steps are supported and decoupled by the component model influences the complexity and the potential for reuse. One important aspect for this thesis is, if the component model allows for substitution of components during operation and if yes, with what impact on the system's availability. If this is not supported, then downtime is required for maintenance and evolution. Another important aspect also mentioned in the component definition by Szyperski is that the composition step can be done by a different organizational unit or even by third parties. Does the component model or platform provide measures to protect the quality of the overall system from problems occurring in third-party components, e.g. prevent illegal communication attempts by containment? If not, this would likely increase the required integration-time efforts due to quality assurance measures. Last but not least we highlight the aspect of performance engineering (cf. [WFP07]): How much model-based analysis of capacity and timeliness is possible and to what degree is measurement-based analysis required late in the engineering process? If much of these efforts happen late (e.g. towards the lower right in a waterfall process), then performance-related optimizations may be required in a try-and-error fashion spanning the complete process and even leading to project failures.

In the last decades, research has been carried out to apply CBSE with the major goal to master the complexity of embedded systems [ABGP05]. In 2005, CBSE had not been widely adopted for embedded systems in lack of runtime technologies fulfilling the demanding non-functional aspects, including real-time requirements [Crn05]. IEC 61131-3 function blocks [IEC13] already allowed for CBSE in industrial automation [ZHD05], though. One of the major advantages of CBSE over other modularization and integration techniques is that all component instances and communication relationships are declared at design time outside of the code. Thus, CBSE has the potential that functional and non-functional integration requirements

can be analyzed at design-time without executing components and even before an implementation exists. Another benefit is that the platform can be just a logical one, or at least that only minimal overhead is caused, by resolving connectors at design time via generators. This was leveraged by the AUTOSAR [AUT16] standard, which has successfully enabled CBSE for distributed control systems in the automotive domain. We elaborate more on specific technologies like AUTOSAR and function blocks in section 2.2.2. Section 2.3.3 elaborates more on formal component frameworks for dynamic reconfiguration.

In chapter 3 we formally specify a component model for modular and evolvable distributed embedded applications first outlined in [TSK18]. Due to our approach, software components have independent life cycles as they are independently developed, built and dynamically orchestrated within containers. Our platform only requires minimal interface and wiring information to manage a software component. One of the primary goals of our runtime architecture is to enforce the component interface – including non-functional aspects – so that the integrator does not need to fully trust the (potentially third-party) component developers, while at the same time, we do not impose many restrictions to the component implementation (e.g., the use of a certain programming language etc.). Moreover, our aim was to make it as easy as possible for the integrator to compose software components and we made it possible to change those compositions even in presence of compatibility-breaking changes and even during operation while maintaining the non-functional (real-time) properties. For these purposes, we proposed reconfiguration extensions in [TK19a] and concepts for enabling state transfer [TK19b], which are formally specified in chapter 4. Our concepts are the first to enable such "arbitrary" dynamic reconfigurations of distributed control systems. In the end, all this work was done with the primary goal to effectively enable agile CBSE for distributed control systems with maximum possible reuse at minimal integration effort, deterministic system reactivity and dynamic reconfiguration at the same time. However, to achieve this breakthrough, we used a minimalistic component model compared to other approaches, e.g. we do not support compositional components. The structure of our component model (PIM/PSM, design-time/runtime, static/reconfiguration, ...) will hopefully help to lift our concepts to more sophisticated component models in future.

2.1.3 Distributed Systems

An important aspect of a component model is how it supports distributed systems. "A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system" [VT17]. Distributing components leads to specific problems: "Reliability, security, heterogeneity, and topology of the network; latency and bandwidth; transport costs; and finally administrative domains" [VT17]. Ignoring them is also referred to as the eight fallacies or pitfalls of distributed systems [Tha09] [Rot06]. In distributed control systems (or: networked embedded real-time systems), these problems may be even more important and difficult at the same time. Based on the definition above, a distributed control system is a collection of multiple autonomous computing elements that collaboratively control a technical process (see figure 2.5). Dedicated design principles for distributed embedded applications exist (the software part of distributed control systems, cf. [Kop11]), especially to achieve deterministic end-to-end timing (see sections 2.2.1 and 2.2.2). In this section we elaborate in general on different approaches to distributed computing to illustrate the required decisions in designing a platform for distributed control systems.

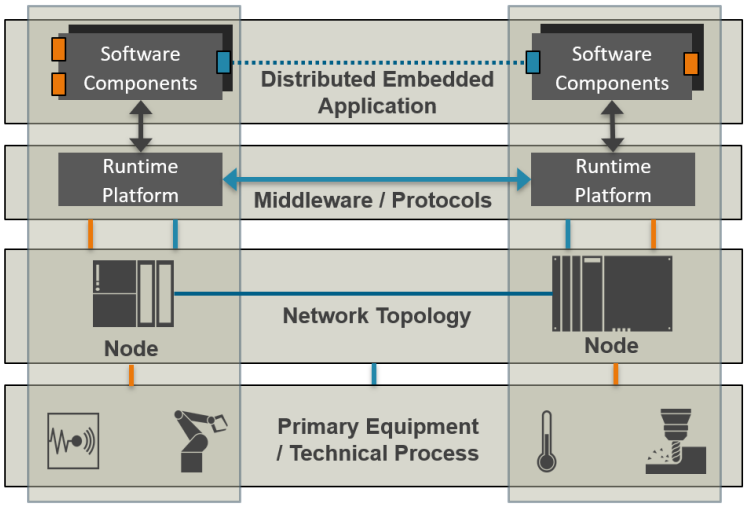

FIGURE 2.5: Schematic overview of component-based distributed control systems.

A common principle in networked systems is to use different kinds of layered architectures, such as protocol stacks according to the OSI Reference Model [Zim80]. In CBSE usually runtime platform and middleware layers exist, which give a strict bound to the ways in which a component can interact with its environment and hides the network topology and deployment. They can also be seen as one abstract platform layer, however, there can be huge differences between different runtime platforms. On one extreme, a runtime platform might not even consider how components communicate with each other, so it can be combined with "any" middleware. The other extreme is that the runtime platform offers platform-specific interfaces for that. The latter approach gives more control to the platform, while the first extreme is more open for application-specific selection of middleware technologies. As a compromise in-between, the platform might support only specific standard communication means (e.g. generic socket-based communication) and introduce some kind of interceptor layer to keep control over the communication. To offer more abstraction and make it easier to stay conformant, client libraries might be provided by the platform for use within software components. They could also be generated based on the component interfaces. The more control the platform has over the communication, the more it can offer abstract solutions for the problems of reliability, security, heterogeneity, and topology. To ensure latency and bandwidth feasibility, the non-functional behavior of components and distributed (embedded) applications need to be declared.

Regardless of the layering, there are different integration styles (cf. enterprise integration patterns [HW04]), which impact the coupling of components. Besides shared memory we can in general distinguish send/receive communication and request/reply communication. Shared memory in its raw form leads to high coupling due to the required co-location and to achieve consistent concurrent access, so we exclude this style for component-based systems. However, storage or database components can be used in the application or platform layer, which strictly translate the memory access to exchanges of messages (e.g. read and write requests) and maintain ACID properties [HR83] as required (e.g. transactional consistency). In a sender/receiver relationship, one component provides values, which are transmitted within messages to the other component if needed, where the values are required. This can be done synchronously and asynchronously. Usually, sending is done asynchronously, so that the sender continues after the value is written, but the sender

might also be blocked or actively wait until the value has been successfully delivered. On receiver side, the platform may trigger the component on reception (event style), or just make new values accessible to the component. In general, synchronous communication leads to temporal coupling of components, at least. Regardless of synchronous or asynchronous communication from component view, the platform may introduce additional delays to transmit and deliver values, e.g. based on time triggers, quota restrictions, or the state of a receiver. Especially message-oriented middleware (MOM) highly decouples the times of sending and receiving, and may, for instance, support an event style reception to be delayed until the receiving component is started. The platform may also support multicasting (1:n) or concasting (n:1) and even blackboard (n:m) relationships, which are particularly common in publish/subscribe-based middleware.

In a request/reply relationship, a component sends a message to another component and expects a reply message. Again, this can be done synchronously or asynchronously. In the synchronous case, the requester would wait for the reply message and only then continue. In the asynchronous case, a callback within the requester might be used, or even polling, while parallel work is continued. Note that the sender/receiver variants of synchronous and asynchronous communication apply both to the request and the reply message. For instance, the request might be stored in a dispatch queue and handed to the receiving component, later, so that only one request is processed at a time. In a client/server form, the reply is provided by the same component to which the message was sent, e.g. in remote procedure calls (RPC). However, especially in MOM-based systems, the decoupling can be further increased by allowing the reply to be sent from a different component after a chain of computations. On requester side the reply is associated with the request based on a correlation identifier within the message either by the platform or within the component. More in general, there may be different integration styles used within the different layers. For instance, the software components might have client/server interfaces, which are realized by the platform via a publish/subscribe-based middleware. Another example is that the platform might provide a (distributed) shared memory interface to the component with synchronous read and write operations, but uses a temporally decoupled message-oriented middleware underneath caring for the synchronization of the data between the components.

An important aspect of the integration styles is how communication relationships are created. In service-oriented architectures (SOA) the client can usually look up the server via its name (e.g. URI) or by some other discovery features. In event-based architectures the publishers and subscribers are coupled by using the same topics, again via the name or more sophisticated features such as wildcard subscriptions. In component-based software engineering, the communication relationships are usually defined by connectors from outside of components. Discovery features may still be supported by the platform – at least in a mocked form. However, it is one of the core concepts of software components that their scope ends at their ports, which should abstract from any concrete relationships with other components. This gives full control to the runtime platform and improves analyzability at engineering time. If communication relationships are to be created dynamically, this should be done via dynamic reconfiguration from outside of components transparently to the inside. It would also be possible to solve this via privileged components or a (self-)reconfiguration interface offered by the platform, though. The communication relationships may also be created by message brokers to abstract the knowledge about where messages need to be sent. A central broker might become a bottleneck,

but there may also be multiple brokers in a logical star topology, or even one broker per node to achieve a decentralized or even peer-to-peer communication on the network level. A broker-less communication can be achieved if the communication relationships are realized without a broker component at all, e.g. by sending or redirecting messages directly between connected ports according to static or dynamic configurations. In the end, these approaches provide different ways to abstract from concrete communication relationships on the lower layers such as different network topologies in the physical layer (ring, star, ...).

Given this wide range of alternative combinations it is a fundamental aspect of this thesis to answer how deterministic end-to-end real-time behavior can be supported by the platform, while enabling dynamic reconfigurations. Our platform-independent concepts aim at making it possible to define a distributed embedded application, and only later define its allocation to nodes. Still the approaches described in sections 3.3 and 4.2 allow that the overall system's behavior can be analyzed formally and deterministically, including the end-to-end reactivity from sensors over processing and communication steps to actuators. Chapter 5 describes a compliant platform realization, which achieves this for UDP-based inter-component communication. This shows that it should be possible to build modular and evolvable distributed control systems based on arbitrary UDP-based middleware technologies. However, some problems are still open for future work, especially regarding some of the mentioned fallacies of distributed systems. For example, we did not include concepts to compensate the loss of a message or detect malicious messages in case of unreliable or insecure networks. These concepts should be possible to add in the application layer based on duplication of messages, checksums etc., but may be solved within the platform layer by future work. Aiming at reconfigurable distributed control systems, our platform concepts do address the latency, bandwidth, topology, and administration aspects, though.

2.2 Industrial Control Systems

An industrial control system uses primary equipment to engage in its environment. The kind and timing of the engagement and its coordination is solved by the software, besides mechanics and the like. It is often important for the purpose of such a system that the end-to-end reactivity from sensors over controllers to actuators is bounded, which leads to real-time requirements. In the sections 2.2.1 and 2.2.2 we elaborate on "classical" concepts and technologies for distributed/real-time systems. More recent approaches are based on more autonomous cyber-physical systems (CPS), which use artificial intelligence technologies to better work in the "real world" without close human supervision [Pla19]. This trend is also called artificial embodied intelligence and predicted to disrupt many industrial sectors and areas of life in the next decade (e.g. smart factories, smart grid, implanted/ingestible microdevices, everyday life robots ...) [BNO+21]. However, the internal implementation as well as the timing and coordination of the engagement of multiple autonomous CPSs lead to additional challenges such as emerging behavior design, runtime system adaptation and scheduling effects of the computing and communication platform [MZ16]. Thus, there is also a trend towards more IT-like technologies on embedded Linux, which provide rich and reusable abstractions that help to solve the challenges of collaborative CPSs. In section 2.2.3 we give an overview of embedded Linux. We also include related "modern" concepts and technologies for distributed/embedded systems within the corresponding sections.

2.2.1 Real-Time Computer Systems

"A real-time computer system is a computer system where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced" [Kop11]. Real-time system consists of one or more real-time computer systems together with the primary equipment and the technical environment. There are soft real-time systems, in which it is acceptable to occasionally miss computation and communication deadlines, e.g. in conditions of a load peak. In this thesis we are aiming at hard real-time, where each missed deadline is critical, as this can lead to total failure of the system and even cause damage. We emphasize that we mean "real hard real-time", where milliseconds matter. We assume that future work will add fault-tolerance concepts so that limited deadline misses can be tolerated (firm real-time). However, the most important aim of real-time computer systems is temporal determinism within a system to contribute to its behavioral determinism within its technical environment. The benefits of determinism (e.g. predictability and composability) and nondeterminism (e.g. simpler abstraction, handling of uncertainty) have been discussed recently by Lee [Lee21]. In real-time systems, determinism is usually valued higher.

We describe fundamental **processing and communication models** for real-time systems based on Kopetz [Kop11]. In general, time-triggered and event-triggered concepts can be distinguished. In time-triggered concepts, processing and communication are initiated at specific (absolute or relative) instants based on clock-related events, usually in specific periods. In event-triggered concepts, processing and communication is initiated by other events such as the arrival of a message. On hardware level, processors and network interfaces are equipped with different kinds of clocks that generate microticks in specific periods of time, e.g. based on crystal oscillations. The microticks can be used to generate ticks (for use as time triggers) and timestamps (for events). Hardware-based event triggers use interrupt mechanisms of the processor to immediately trigger an action, e.g. when a new message is available in the buffer. However, the processing and communication models on the software level usually provide abstractions of the triggering of the hardware. These abstractions should separate the two concerns of determining the instants at which application parts and communication should be triggered (temporal control) from the logic and interactions of the time-consuming application parts (logical control) [Kop11]. The more directly the logic can influence the processing and network schedule (e.g. by raising an event or by sending a procedure call request to instantly execute logic in another application part), the more difficult if not impossible it becomes to guarantee for hard real-time properties of such a system.

A real-time application is often structured in independently executable sequential programs called tasks, which are managed by some kind of real-time operating system (RTOS). The RTOS controls the **CPU scheduling** of the local tasks, e.g. via execution control messages or by simply calling tasks as subroutines (especially for generative RTOSs). Once executing, a task computes outputs from inputs, potentially using and updating an internal state, which summarizes information about the past. Tasks interact with other tasks via an application programming interface of the RTOS, e.g. queues. To increase dependability and support "reintegration" (e.g. replication) the state of a component can be externalized between cycles (ground state), potentially using a database component. A fixed CPU schedule may be calculated at design time, or the schedule may also be dynamically managed by a scheduler. Dynamically scheduled tasks are preempted by a scheduler of the RTOS, e.g. when a task with higher priority becomes ready. To guarantee for hard real-time

properties the schedulability of possible task sets must be analyzed at design time. For completely time-triggered (periodic) task sets, the schedulability can be decided based on the tasks' periods, deadlines and worst-case execution times. Only one schedule period needs to be considered (the least common multiple of all periods), because the schedule can be repeated. The analysis depends on the scheduling algorithm, though. As a generic necessary schedulability test, a task set is not schedulable if more execution time is required than provided, i.e., if $\sum c_i/p_i > n$, with n processors, periods p_i and execution times c_i. However, there may also be event-triggered (sporadic/aperiodic) tasks, for which we do not know apriori the times at which execution is requested. They can be handled statically by treating them as periodic tasks based on a minimum request interval, e.g. by sporadic server tasks. Alternatively, sporadic tasks can also be treated by mode changes, i.e., changing to a different static task set and schedule in case of an event. For independent tasks (no precedence/order, no semaphores etc.), dynamic scheduling algorithms such as rate-monotonic scheduling (RMS) and earliest deadline first (EDF) can be used. In RMS, the tasks with the shortest periods (assumed to be equal to their deadlines) are assigned the highest static priority. If the utilization does not exceed $n(2^{1/n} - 1) \geq 0.693$ (for n tasks), RMS guarantees all deadlines by always executing the highest-priority task ready to run [LSD89]. A task set with utilization of 1 is schedulable if the periods are harmonic. With EDF, a utilization of 1 can be reached without harmonic periods by always selecting the task with the shortest time left until its deadline [LL73]. RMS and EDF are optimal for single-core systems, i.e., they always fulfill all deadlines if the task sets are schedulable, but not for multi-core systems. They are also combined with server task models such as constant bandwidth servers [AB04], especially to isolate soft or non-real-time tasks in mixed-criticality systems [BD13]. Additionally, they are combined with statistical approaches to reduce the required over-provisioning by utilizing variability of the execution times, e.g. statistical RMS [AB98]. However, in case of dependent tasks, a higher priority task might have to wait for a lower priority task to free a shared resource or provide a result (priority inversion). To solve priority inversion and avoid deadlocks, the priority ceiling protocol can be used on a single-core system, if all resource requirements of the tasks are known apriori. Each resource gets the priority of the highest-priority task that may request it. On requesting a resource, a task is not only blocked if the resource is occupied, but also if the task's current priority is lower than any occupied resource's priority (the priority ceiling). While accessing a resource a task dynamically inherits higher priorities from any blocked tasks. This makes sure that resources are not allocated in a way that prevents tasks from making progress due to circular resource blocking. At the same time, higher priority tasks are always favored except if they are blocked by lower-priority tasks. Thus, the schedulability for each priority can be analyzed based on the utilizations of the tasks with higher or equal priority and the worst-case blocking times by lower-priority tasks. However, the exact timing behavior related to the dependencies must be known and analyzed, i.e., we also have to consider consistency of the logical control in the schedule, as the tasks usually exchange results.

These efforts to solve temporal control consistently with logical control stand in contrast to our intention to easily and even dynamically integrate software components in a CBSE approach. Therefore, we seek for ways to simplify or completely **abstract from scheduling** and find it in the LET paradigm [KS12]. In the LET school, the RTOS approaches above are classified as Bounded Execution Time (BET) abstraction, with the period and deadline as the bounds of the execution. From a mathematical point of view BET-systems are in general usually modeled as Timed

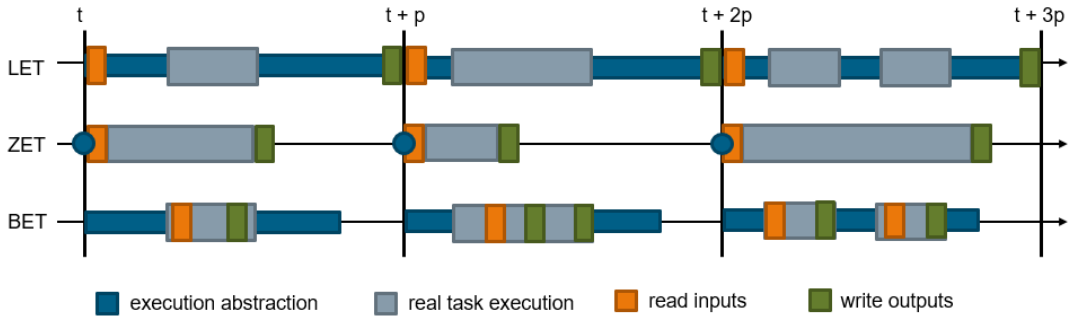

FIGURE 2.6: Overview of the three fundamental real-time programming abstractions (cf. [KS12] fig. 1). LET seems to be the most modular and evolvable abstraction for distributed control systems due to the strong temporal decoupling.

Automata [AD94]. Between BET and LET, the Zero-Execution-Time (ZET) abstraction exists, which abstracts from the execution time by the assumption that a task terminates instantaneously, including input and output processing. The ZET approach has been leveraged by synchronous languages such as Esterel [Ber00] and LUSTRE [HCRP91], which are compiled to one big linear task to periodically calculate a logically instantaneous transition of a Mealy machine [Mea55] defined by the program [Ber00]. To ensure correctness and hard real-time in particular, fixed-point analysis is used to prove that the program terminates (before the next period starts) [KS12]. The LET paradigm is a combination of BET and ZET, which was first proposed for Giotto [HHK03]. Figure 2.6 shows an overview of the three abstractions. It abstracts from the execution time by assuming that the task starts executing at the start of the period reading the inputs exactly at this instant (under ZET assumption) and terminates exactly at the end of the period providing the outputs (under ZET assumption, again). This can be achieved logically by providing the inputs and taking the outputs only between task executions, keeping back any new inputs our outputs until the subsequent logical cycle turnover. As the execution platform processes the inputs and outputs in the cycle turnover, it must be done under ZET assumption, again, as there is no time for this, logically. With the LET paradigm, we can more easily assign more concurrent, independent tasks, compared to BET, as the inputs and outputs are "frozen" during task execution, which decouples the schedule to the extent that only the processor utilization needs to be considered. Compared to ZET, the "frozen" semantic has the additional benefit of temporal determinism with regard to the technical process. The temporal decoupling of tasks and the resulting simplicity to analyze the end-to-end reactivity are major benefits of the LET paradigm [HDK+17]. This decoupled and deterministic processing chain costs more runtime management overhead and more delay of the computation, though. From an architectural perspective, this means that I/O and communication has to be managed and to some degree implemented in the platform layer.

The benefits of simplifying scheduling become even more important if we consider chains of computations by **distributed tasks** and the communication between them. For minimizing the end-to-end reactivity of a real-time system, these chains could be phase-aligned by a holistic schedule, which plans a time trigger for the next task activation or message transmission just after the preceding step is known to

have terminated as of the worst-case execution time (including any potential blocking) or worst-case communication time. The communication scheduling must therefore be considered in such a holistic schedule. The physical communication scheduling might be based on static time slots (e.g. CAN, FlexRay, EtherCAT) or allow for dynamic network transmission (e.g. Ethernet, Wi-Fi). On top of this, the protocol stack and the platform could provide abstractions for the network scheduling, for example, based on queueing disciplines, but it might also allow to directly write into a time-triggered protocol frame, which thus leads to strong coupling of the task and network scheduling. Depending on how the communication should be aligned with the schedule or how events are ordered, the involved nodes might also have to synchronize their clocks, e.g. by specific protocols such as PTP [IMS19]. For the three different abstractions, following general concepts were proposed to minimize the complexity of distributed scheduling. For classic BET-style systems, the Time-Triggered Architecture (TTA) [Kop11] was proposed. In essence, all periodic task executions and communication channels are timed in a rigid global schedule based on an established global time base. Additional dependability concepts such as component replication and voting, periodically persisting a ground state, and even a secure download service (no hotfixing, though) are described as well. If the principles of TTA are followed, Periodic Finite State Machines (PFSM) [OEHK07] can be used to model and analyze the behavior of a distributed control system. For ZET-style systems, the PALS concept (Physically Asynchronous Logically Synchronous systems) [SAS+09] can be used to achieve synchronous distributed systems. The idea is to ensure that outputs of cycle n will be processed during cycle $n+1$ also on remote nodes. For this they must be buffered during cycle n on the receiver end, of course. As clocks are not perfectly synchronous, messages must be sent only in a time interval distant enough from the cycle start (by delaying if needed) and end (by tuning period and WCET) depending to the bound of the clock skew. For distributed systems, LET can be expanded to the network communication, as we proposed in our Real-Time Container Architecture [TSK18]. Our architecture solves this reconfigurably on the platform level by an agent and a special *barrier* queueing discipline as described in the subsequent chapters. From abstraction perspective, the same approach was also proposed as System Level LET, to achieve better analyzable distributed systems with regard to time-sensitive cause-effect chains [EAG18]. In System Level LET, communication tasks (could be application- or platform-level) make sure that an output from the previous cycle intended for a remote receiver is transmitted exactly during a period, and that all tasks on all nodes are executed in globally synchronized isochronous cycles. This simplifies the end-to-end analysis, as each processing and remote communication step takes exactly one logical period so they are chained temporally deterministic. The chain analysis can be done as described by Hamann et. al. [HDK+17] or based on dataflow graphs using our formal model as described in section 3.3. The LET paradigm is also the basis for reconfigurations that must be executed synchronously across nodes to maintain system consistency and availability, as we proposed in [TK19a] and also elaborate in the subsequent chapters. For analyzing the end-to-end behavior during concurrent reconfigurations, Evolving Dataflow Graphs (EDFG) can be used as proposed in section 4.2.

Thus, the LET paradigm seems to be the most modular and evolvable processing model for distributed control systems. This can outweigh the latencies caused by the logical execution time and logical communication time, if the resulting reactivity is still sufficient. The LET paradigm is the enabling basis of our concepts, including our CBSE and MDE approach, the container-based execution platform concept and

the reconfiguration model. The combined concepts separate multiple concerns as elaborated in [TK19a]. The most important one is to separate the scheduling of tasks and messages from the functionality and contents of tasks and messages. This way we can reconfigure consistently and reactively without involving the application in reconfiguration management. However, it might be possible to apply our concepts to TTA and synchronous distributed systems in future work, as these architectures also enable deterministic end-to-end chains with low coupling.

2.2.2 Distributed Control and Communication Technologies

We describe concrete related platform technologies for distributed control and communication, including technical standards and selected proprietary platforms, which we are aware of. The technologies combine different (non-/real-time) computing concepts and solutions for distributed (non-/real-time) communication. We also include the aspects of MDE and CBSE, as well as features for dynamic reconfiguration.

Automotive

A typical and widely used RTOS standard is **OSEK-OS** [ISO05a], especially used in automotive control systems. It requires fixed priority preemptive task scheduling with priority ceiling, though non-preemptive, mixed preemptive and full preemptive are possible, depending on the task set. Basic tasks run to completion unless an interrupt occurs or they are preempted by a task with higher priority. Extended tasks can additionally request to wait for an event, so that they release the CPU for lower-priority tasks and only become ready again when the event is set. An event is a binary information assigned to one task, only, so only the owning task can read and reset an event flag or wait for it to be set. The event can be set from any other tasks and by some interrupt service routines (ISR). There are two ISR categories, with and without OS calls, which can be suspended all together or only the ISRs with OS calls. If an ISR sets an event via the OS call, the tasks will be newly scheduled after all ISRs have completed. The OS also controls concurrent resource access between tasks and ISRs via critical sections (scheduler, code, memory, or hardware). Thus, OSEK-OS is an event-triggered RTOS with the full scheduling complexity pointed out in the previous section. However, besides events set by programs, there are also alarms, which enable dynamically configured clock counter condition triggers (i.e., time triggers). At least one counter exists, the timer, for which a counter value condition can be configured to periodically execute tasks or alarm callbacks. Thus, OSEK-OS can be used to define time-triggered real-time systems, too. The OSEK-COM [ISO05b] extension of the standard defines an API for communication between tasks and/or ISRs – including remote tasks. It hides from the tasks the details of the deployment, the bus technology, and the network topology to increase portability, reusability and interoperability, but still aims at hard real-time communication. The interaction layer of the OSEK-COM specification requires that all messages and their metadata are statically configured, so that corresponding functions for sending and receiving them can be generated. Internal and external message objects are then written to or from message objects, which may have either queued (fixed queue length) or most-recent behavior. The message objects also support abstractions to filters (time/value) and to monitor whether deadlines were kept (both on sender and receiver side). The latter can be configured with different mechanisms including setting a flag on the message object, setting an event of the task, or activating an error task. Internal messages (between co-located tasks) are delivered immediately, i.e., copied to the

message objects on the receiving end (potentially, multiple ones). External messages are delivered indirectly via protocol data units in the network layer. While writing the data to the corresponding frames, offset and endianness are handled according to the communication configuration (in particular, the size of each message is fixed). The network layer also issues transmission requests and reacts on reception indications in its interaction with the data link layer (which is not further specified in the standard). Both direct transmission mode (event-triggered transmission by sending a message by the application) and periodic transmission modes (time triggered) can be supported, as well as mixed transmission mode. On the receiving end, the network layer extracts received messages from the protocol data units to the configured message objects. As all this functionality is generated based on a configuration, the OSEK-OS configuration is a complex task, which highly influences the end-to-end timing analysis.

On top of OSEK-OS, the **AUTOSAR** [AUT16] standard (meanwhile called AUTOSAR Classic Platform) has been defined and successfully enables CBSE for distributed control systems in the automotive domain. It defines a component model and runtime environment so that software components could be logically wired via a virtual functional bus (VFB) before defining the allocation to hardware in a later engineering step. The runtime environment can be seen as a component-based configuration abstraction of OSEK-OS, and the virtual functional bus as a component-based configuration abstraction of OSEK-COM. A software component has multiple tasks (behavior) with different sorts of triggers (time, messages, ...) and ports with sender/receiver interfaces and client/server interfaces, which are translated to OSEK-OS features. For instance, a remote procedure call behavior is realized on server side by an OperationInvokedEvent that triggers the task execution. A periodic behavior is achieved by a TimingEvent, which is realized by a timer-triggered alarm. In addition, the runtime environment standard includes basic software modules to further standardize the configuration of the systems, e.g. the data link layer is standardized (CAN, FlexRay, ...). Generator tool chains then create a monolithic image for each Electronic Control Unit (ECU) based on extracts of the system description, a configuration of the platform, and the component interfaces and implementations, so that for each ECU a different vendor's toolchain and runtime environment could be used. Several modeling and configuration formats have been defined for AUTOSAR, for example ARXML as core of AUTOSAR (cf. [AUT19a]), ARTEXT [Art22] on top of ARXML, and EAST-ADL [CFJ+10; Ass13] for supplementary modeling, e.g. feature and safety requirements, including timing constraints. However, despite the wide tool support for these formats by various vendors, the configuration of the platform is complex, as many low-level details have to be provided by the engineer, for example on protocol data units, OS tasks and I/Os. This makes even static reconfiguration a big effort, but dynamic reconfiguration is very limited due to the generative, monolithic architecture of AUTOSAR classic. The LET paradigm has also been proposed on top of the AUTOSAR classic platform to allow for deterministic multi-core execution [BD18]. A similar approach was also described by [HDK+17], who describe the end-to-end analysis. Meanwhile, LET is even part of the AUTOSAR standard [AUT19c]. This can simplify integration and configuration, though at least tampering with configuration items such as *PeriodicEventTriggering* and *OffsetTimingConstraint* is still possible and to some extent required. However, dynamic loading of components and system modes would be required in addition to the LET paradigm to apply dynamic reconfigurations as proposed in this thesis. Another direction to decrease complexity is constraint-based system synthesis [GKC+15], which was also used in a concept

to enable pre-defined dynamic reconfiguration of AUTOSAR-based distributed control systems [SDWB17]. This concept can mitigate failures of control units by a monitoring and reconfiguration service, which triggers failure-specific mode switches to degraded alternative architectures when periodic remote heartbeats are missing. Those alternatives are derived from quality declarations that allow to automatically synthesize "second-best" systems for each failure. These dynamic reconfigurations are statically pre-defined and applied only locally, so they do not allow to roll out new functionality or to break compatibility across nodes, though. In this sense, dynamic reconfiguration of distributed embedded applications is still not possible in the AUTOSAR Classic Platform.

In the more recent **AUTOSAR Adaptive** Platform [AUT21b], this standard now moves on from a static platform to a more dynamic one, in which software components and relationships between them are created at runtime. Instead of OSEK-OS, adaptive AUTOSAR is based on POSIX [ICS17]. We describe some related details on POSIX in the Linux-related section 2.2.3. The AUTOSAR Adaptive Platform standardizes how such systems should be initialized, what functional clusters and AUTOSAR services should be present, and the communication interfaces between them as well as the configuration of the OS to run applications (i.e., POSIX processes). For instance, the platform configures resource groups for the applications. In general, platform- and user-level applications have the same access level – so it is the responsibility of application developers not to access OS services and APIs in non-compliant ways that restrict the portability of other applications. Processes can and must request real-time scheduling policies themselves via the OS interface if needed, but several settings should be configurable via a manifest file (priority, CPU quota and affinity, RAM limits, ...). The *DeterministicClient* platform API offers a service *WaitForActivation* for cyclic task execution, which blocks until the next cycle shall be triggered. The communication management specification [AUT21a] covers aspects such as end-to-end communication protection and inter-application communication. For instance, inter-application communication could be realized via DDS, for which a network binding is included. The current specification does not describe how deterministic synchronization across nodes should be done, though it is specified as a requirement. Execution and communication management is done during operation, but only during system start to achieve a more modularized deployment, while updating of processes during operation is not covered in the standard. The standard is still emerging, but up to now it does not provide end-to-end temporal determinism. Menard et al. proposed the reactor-based solution DEAR [MGLC20] on top of AUTOSAR adaptive to ensure logically deterministic (order-preserving) execution of concurrent chains of computations without strict scheduling. The reactor concept enables this by associating messages (related to method calls or events in AUTOSAR adaptive) with a logical time called tag. On the start of a processing chain, e.g. triggered by sensors, the physical time is used, whereas the consequent messages keep the same tag. Based on the tags reactors then delay execution of actions (that react on input messages) according to a safe-to-process analysis, which takes into account an upper bound of the clock skew and communication latencies, but also deadlines, so that the order of message processing is preserved despite non-deterministic dispatching on runtime level. Due to this concept, the reaction chains are ordered consistently and logically instantaneous, but unlike the LET paradigm the composition of reactors does not require explicit temporal alignment, which is difficult to apply in AUTOSAR adaptive [MGLC20]. However, even though it was

claimed that reactors facilitate analysis of end-to-end latencies [MGLC20], the problem remains that it is difficult to analyze statically at design time, what the worst-case communication time and effective computational reaction time will be, as the number of parallel reactor executions and message transmissions as well as their impact on the scheduling is not known. Finally, we are not aware of approaches for reactive and deterministic distributed reconfiguration of such systems.

Avionics

In avionics, the ARINC 653 RTOS standard is the basis for safety-critical and real-time distributed control systems [Pri08]. The scope of ARINC 653 is similar to the scope of OSEK-OS. It includes task management based on partitions, the processes of which may be time-triggered and event-triggered, which however is done in isolated fixed time slices based on process scheduling attributes to ensure temporal determinism. Besides the CPU time, the memory regions are partitioned to ensure the isolation of the partitions via memory management features of the hardware. For inter-partition communication, ARINC 653 includes a queue-based communication abstraction, which also abstracts from the deployment of partitions. Synchronous and asynchronous communication is supported, with polling or completion alerts. It can also be used to achieve client-server communication, even dynamically via service access points. Using sampling ports, the communication can be done in typical data structures used in aviation data busses, e.g. including a status indication [Pri08]. Mode management is supported, too, so that different static task sets can be defined. I/O management is done by the RTOS, including fault monitoring and triggering of specific handling functions. The resulting time and space partitioning of ARING 653 is used to achieve Integrated Modular Avionics (IMA), i.e., CBSE in avionic distributed control systems [GWC15]. To achieve distributed determinism as a prerequisite, the Time Triggered Architecture (TTA) [Kop11] was also proposed for avionic systems in three layers: to use Simulink on design and simulation level, translate to Lustre to leverage the DO178B-level-A automatic code generator, and integrate into a distributed system according to the Time Triggered Architecture (TTA) using the time-triggered ARINC 653 RTOS on each node [CCM+03]. Time-triggered Ethernet with time synchronization and time division multiple access (TDMA) was used to partition the network in the same way as ARINC 653 partitions the CPU and memory, e.g. in three traffic classes [GWC15]. However, quality-of-service-based distributed communication has also been used in avionic standards. As IMA is about defining a complete airplane architecture, the integration of the individual subsystems and components is highly complex. It results from configuring CPU, memory, and network resources as well as communication relationships while ensuring end-to-end reactivity. For this purpose, interface control documents (ICD) have been extended so that the information required for the corresponding IMA concepts were included [GWC15]. There is ongoing work to harmonize ARINC 653 and POSIX to fulfill the FACE requirements, an emerging military avionics standard [BSG18] [BS20]. This leverages hypervisor features of an existing ARINC 653 operating system to run POSIX-based guest systems. It was even claimed that automotive and avionic technologies may have synergies, e.g. in a unified hypervisor infrastructure [GWC15]. An older approach still relevant to this thesis is Giotto [HHK03], because this was the birth of the LET paradigm described earlier. A Giotto program essentially consists of tasks that have input and output ports (data points in a global space) and private ports. Additionally, mode switches are possible to change the task set. The tasks are invoked at isochronous instants

according to the LET paradigm, i.e., such that input valuations are only changed at the cycle start, and output valuations are ignored until the cycle end. The logical execution time of a task is determined from the current mode's period (e.g. 10 Hz) and the task's frequency (e.g. 5, leading to 50 Hz invocation frequency or 20 ms logical execution time). Only at cycle turnovers, the platform invokes drivers to provide declared inputs from sensors and outputs to actuators, which is logically done in zero time (ZET abstraction). Giotto includes a DSL for describing the system architecture, from which a schedule description can be generated – the *E code*. The platform-independent *E code* is finally used by the *Embedded Machine* at runtime to schedule the tasks and drivers accordingly using platform-specific features of the underlying RTOS. A well-timed Giotto program is schedulable for given WCETs of the tasks if for all mode task sets M the sum of all individual task utilizations is smaller than or equal to one [HKMM02]:

$$\sum_{t \in M} WCET_t / LET_t \leq 1$$

A Giotto program is well-timed if mode switches only happen at the end of a mode or otherwise if all tasks scheduled during a potential mode-disrupting switch are present in the target mode as well and have the same LET. Hence, schedulability analysis is highly simplified with the LET abstraction in Giotto. The HTL [HKMS09] language and framework adds modules to the Giotto concepts, i.e., components. Modules in HTL have modes and tasks, and communicate with each other via a set of communicators. Modules are distributed to a set of hosts connected by a "reliable, time-synchronized broadcast network" [HKMS09], so communicators also care for remote communication. Hierarchical refinement is possible by declaring abstract tasks in a mode. The concrete tasks are defined during mode switches (e.g. selection of a fast or slow implementation). A little generalization of the Giotto task model was proposed for that. The logical execution time of a refinement must be larger than of the abstract task (though we believe that breaks the Liskov substitution principle [Bar88]). To cope with that, inputs can be provided also before the logical release time of a task, and the outputs can be released after the logical termination time. Based on this platform, a concept for model-preserving runtime patches [KLMS11] was proposed. A runtime patcher replaces a module, mode, or refinement task with a new one during an atomic switch. At this time, all tasks of the module must be in a quiescent state regarding communicators, and the behavior with regard to communicators must stay compatible. This is ensured by analyzing the program and the patch at design time and instructing the runtime patcher with the new and intermediate *E code*. This schedule will, for instance, temporarily block processes if needed, and activate different tasks at certain points in a period. However, still none of these approaches aim at compatibility-breaking dynamic and deterministic reconfiguration of distributed real-time systems during full operation. Finally, we refer to a CBSE approach for avionic systems described by Panunzio and Vardanega [PV14], which is very similar to our envisioned engineering process described in chapter 3. This approach describes decoupled creation and reuse of components, their assembly, their deployment, to then running the modeled system by a generated real-time and communication architecture. In addition, our runtime platform concepts provide the possibility to do each of the decoupled activities iteratively and apply changes easily, even during operation and even in case of breaking changes, by our means for synchronous reconfiguration during operation.

Automation

We already gave an overview on trends and technologies in manufacturing in the introduction. In this section, we elaborate more on related runtime platform concepts for distributed control and communication technologies. In industrial automation, usually PLCs [Bol15] are used to run **control logic**. A PLC is a special-purpose microcontroller, which continuously runs a program cycle. PLC programs are usually defined via programming languages defined in IEC 61311-3 [IEC13]. For instance sequential function charts (SFC) can be used to specify a cyclic procedural state machine, which is basically a Petri-net [MM21]. The individual functions and aggregations of them that make up a PLC program are often generated from complex mathematical models (e.g. integer programming) that are beyond the scope of this thesis. We refer to Williams [Wil13] for more background on building such models. A PLC works as follows [Eri96]. At the start of a cycle, the inputs are copied from the input modules to the memory. Then a sequential program is executed, which takes the copied inputs and its state and updates its state and internal outputs (i.e., it is a Mealy machine). Only after it completes, the internal outputs are propagated to the output modules. Usually the next cycle starts immediately afterwards to improve the reaction time, so the cycle time is not a constant period. In addition to varying execution times, interrupts may be used to trigger intermediate logic, pausing the cyclic execution. While this is intended for quick, event-triggered execution, it can also be used to achieve a time-triggered execution via timer interrupts. Thus, PLC-based automation can also reach the full real-time analysis complexity as described for RTOSs. For instance, different timers with rate-monotonic priorities could be defined to which the different application parts are mapped, which leads to the RMS utilization issues mentioned before. However, this could also be used to define completely time-triggered systems, e.g. according to the LET paradigm, because the freezing of inputs and outputs is naturally provided by the PLCs. Additional complexity arises from advances on the hardware level. While traditionally, special-purpose PLCs where used as reliable hardware platform, meanwhile also Industrial PCs (IPCs) (standard PCs with emulated PLC environments) are more and more popular due to hardening of the standard hardware and OS, possibly even including a real-time separation kernel [Wil15]. This trend is recently expanded towards edge control, where hypervisors are used to run real-time control side-by-side with arbitrarily rebootable general-purpose operating systems, which are used to run edge and IoT functionality (see also section 2.2.3) [Tho22]. This is thus a similar direction as in the avionic and automotive domains described before. With our design-time, runtime and reconfiguration concepts in this thesis, we are aiming at the real-time parts of such systems, however.

Communication between PLCs can be done by several industrial standards, including OPC UA [ISO20], the field-level communication standard Profinet [IEC20], but also DDS [OMG15a], more recently. These interoperability standards address the need to integrate machines and controllers from different specialized vendors for real-time control. The integration styles range from client/server in OPC UA (classically) over cyclic sender/receiver in Profinet to Publish/Subscribe in DDS. Traditionally, the timing of the distributed communication is not fixed, as often only the most-recent value is important. Real-time control can be supported by prioritizing communication and time-triggered traffic scheduling, e.g. within the protocol (as in Profinet) or by configuration of the bridge layer [ICS18]. Consequently, the end-to-end behavior is not automatically deterministic, so depending on how tasks and communication are triggered it can be difficult to analyze the end-to-end timing

behavior of a distributed PLC-based control system. IEC 61499 (distributed function block diagrams) [IEC12] is a standard for distributed control systems based on PLCs. IEC 61499 applications consist of (compositional) function blocks (FB) that communicate with each other via events and data connections. The system configuration maps FBs to devices and inter-FB communication to protocol messages – potentially in a subsequent step following their wiring [DV07]. Due to the holistic architecture model that includes the full deployment, end-to-end analysis is possible, provided that the runtime and communication platform are selected and configured accordingly. Thus, IEC 61499 is a good basis for CBSE of distributed control systems, for instance with a V-Model methodology proposed by Zhang et al. [ZHD05]. On top of that, IEC 61499 supports dynamic reconfiguration as described later in this section.

Unlike in automation and avionics, the distributed control system needs to be changed more often and more extensively in automation. The importance to design **reconfigurable manufacturing systems** was described by Koren et al. [KS10]. According to them, manufacturing systems need to be reconfigured in case of market demand changes, product changes and system failures, which can all happen frequently. While their focus is on how to organize the machines and logistics to achieve reconfigurable manufacturing lines, they also mentioned changing the software structure as a means (downtime is assumed, though), leading to the requirement to use advanced controllers. They propose six principles of reconfigurability: customization, convertibility, scalability, modularity, integrability, diagnosability. These can also be applied to software reconfiguration. The software components should be customizable via configuration data and convertible in the sense that they can be reused in different situations. Scalability is needed so that we can change the number of nodes and software components. A modular and easy to integrate architecture is needed (which leads us to CBSE). And finally, the system and reconfiguration status needs to be diagnosable. We add two interdependent aspects to the six principles: coordination and dynamism. First, collaborative manufacturing systems are more complex than just serial and parallel lines of machines, as the machines could dynamically create collaboration relationships based on their internal behavior and might even require adaptation of this behavior for concrete collaborations. Second, if we can dynamically reconfigure on software level without downtime, there might be no line changeover time at all or it could be done deterministically in short idle moments. As mentioned in the introduction, much research is going in the direction of process level reconfiguration. However, we think that the concerns of deterministic dynamic distributed software reconfiguration and production line reconfiguration should be separated into two reconfiguration layers as far as possible, as the complexity of both aspects is high. For instance, if you consider autonomous machines interacting via some shared language or interaction conventions, it would probably be too complex if arbitrary dynamic software reconfiguration had to be incorporated in the design of these interactions, too. Therefore, the process level reconfiguration is not considered in this thesis, at least not beyond analyzing the system behavior also during reconfigurations. Instead, we provide a dynamic and deterministic distributed reconfiguration approach, which can be used by additional concepts on top to achieve dynamically reconfigurable manufacturing systems. These concepts will probably require holistic, integrated reconfiguration on the two layers, the coupling of which can be minimized, though, if the software reconfiguration layer provides determinism. Therefore, this thesis focuses on modularity and integrability to facilitate CBSE as well as design-time and runtime concepts to achieve analyzable dynamic reconfigurations with temporal determinism.

We describe existing approaches to **dynamic reconfiguration** of PLC-like control

systems. Wahler et al. [WRO09] proposed a concept for updating individual cyclic components, which takes into account the timing of the update, a general algorithm (download and initiate, exit old version, handover state, acquire and start new version) and especially real-time state transfer. As of this concept, a state conversion function should be part of the new version and executed just before the first cycle – it is assumed to fit into that cycle (which worked up to 300 Hz in their evaluation). If it does not fit into the cycle, they mention (but do not solve) the problem of maintaining ACID properties during simultaneous replacements of multiple components. This problem is even bigger in distributed and compatibility-breaking reconfigurations, which we address with our concepts in this thesis. A prerequisite for transactional consistency of such reconfigurations is determinism of the changes within the end-to-end dataflow, which we achieve by our concepts. A more recent approach for live updates in PLCs was proposed for Cetratus [MM21]. The concept is based on a switch between a primary and a secondary container for Petri-net logic (i.e., SFC). An update package contains the new Petri-net model (i.e., the new program) and a state transformation matrix. The reconfiguration is triggered by an update request, after which the new Petri-net model is loaded in the secondary container. Additionally, the state of the primary model is loaded and potentially transformed using the state transformation matrix within the first execution cycle in which the secondary Petri-net is also computing the state equation, but its outputs are only monitored, not yet provided to the equipment. This leverages their I/O virtualized environment Cetarus, in which the inputs and outputs of a program are not directly exchanged with the equipment, so the secondary container can be muted in order to perform an in-field pre-activation test. The user monitors the behavior of the new version and trigger the atomic switch, so that the container with the new program becomes primary and its outputs are used instead. While the evaluation example is a reconfigurable Vernadat system consisting of two machines and a robot, the Petri-net is not distributed, not even over multiple containers on the same machine. An update approach for IEC 61499 [IEC12]-based distributed control systems was described by Zoitl et. al. [ZLMV10]. The IEC 61499 runtime should provide basic reconfiguration services, which are meanwhile part of the standard. Using these services, a reconfiguration application can create and delete structural elements such as function blocks and manipulate their states. Similar reconfiguration features can be found among our modification steps and lifecycle steps (see section 4.1). Though the reconfiguration application may manipulate the distributed control system, reconfigurations are restricted to having only local effects, i.e., no compatibility-breaking changes are possible in lack of a concept to ensure distributed integrity. This is addressed by our runtime and engineering concepts as described in the subsequent chapters (e.g. by distributed LET, coordination steps, Evolving Dataflow Graph ...). Another approach to dynamic reconfiguration of IEC 61499 systems was proposed by Prenzel and Provost [PP17]. They transform a function block diagram to Erlang OTP, which natively supports updating of individual processes. Each process runs the Moore machine of one function block, converted to Mealy machines as supported by Erlang OTP, and communicate via Erlang events in an actor style. The new code modules are started, the processes to update are suspended, state transfer is performed, and then the processes resume in the new version. The updates needed to be backwards-compatible, because distributed dependencies were not considered. Additionally, there is a non-deterministic downtime of the components. The concept was extended by Prenzel and Steinhorst [PS21] so that reconfiguration sequences of reconfiguration operations defined by the IEC 61499 reconfiguration services were

generated automatically. The ordering of start and stop operations in case of simultaneous function block updates is solved by topological sorting of a dependency graph. Additional state transfer operations are included between stopping an old version and starting a new version, potentially using a user-defined function block for state transformation. The reconfiguration operations are strictly ordered based on constraints, so that the consistency of the application can be guaranteed in the sense of an interleaved switch from the old to the new event chain. Recently, Prenzel et al. [PHS22] also added the possibility to quickly rollback dynamic reconfigurations in case of a failure. The concept is based on the observation that most reconfiguration operations are reversible (for instance, creating a function block can be undone by deleting it, again), while only few of them lead to a point of no (quick) return (e.g. deletion of a function block). The reconfiguration can thus be rolled back in-time by applying the inverse operations in reverse order, if the fault happens before the point of no return. However, the downtime during state transferal still remains and sums up along the event chain. This limits the applicability of the reconfiguration concept to distributed function block networks, in addition to the fact that remote communication with reconfiguration services is needed in time-critical reconfiguration phases to ensure a consistent timing of reconfiguration operations. However, as some of the reconfiguration concepts are similar to our approaches, it may be possible to introduce an adapted form of our distributed coordination concepts to reduce downtime. This would require more temporal determinism for this event-based runtime, e.g. by a reactor approach as it was proposed for AUTOSAR adaptive [MGLC20] (see also section 2.2.2).

2.2.3 Embedded Linux

POSIX [ICS17] is a widely used OS standard in desktop and server systems, but also in embedded systems, as it also includes real-time scheduling interfaces. It is implemented by many RTOSs, though not all of them are fully compliant and certified. For this thesis, Linux is most relevant, as we built our evaluation prototype on a custom Linux distribution (see chapter 5). Many Linux distributions can be used as RTOSs, too, especially with the real-time patches applied. This is also referred to as embedded Linux. The major benefit of embedded Linux is the wide range of mature software, including drivers and secure virtualization and network stack components, and its customizability. To create a customized Linux distribution, especially in embedded domains, the Yocto build system can be used, which we also did for our real-time container architecture prototype. Our prototype concept relies on some CPU and network scheduling features provided by Linux, and especially some containerization features.

Linux provides most system functions, subroutines, shell and utility programs as specified by POSIX. This includes a portable character set, locales, tools (e.g. *crontab*, *find*, *patch* ...) and functions (*pthread_create*, *timer_create*, *mmap* ...), as well as the file system hierarchy (temporary files under `/tmp`, devices under `dev` etc.). However, we are more after the general concepts behind the interfaces related to the core OS features, especially those related to embedded systems and containers, some of which are beyond POSIX. In general, application programs consist of one or more processes. Processes may also contain threads, which are treated like processes, except they share the same address space. Linux separates processes and threads in user space from lower-level functionality in the privileged kernel space (including kernel modules), e.g. scheduling and I/O. When a user space process requests services by the kernel by system calls, the kernel checks whether the process may do so and what

information is visible based on several mechanisms. Whether specific functions and features are accessible to a process at all is controlled by non-standardized privileges, called capabilities in Linux [Ker02]. For instance, `CAP_SYS_CHROOT` is required for calling *chroot()*, which can be used to lock a process into a fake root filesystem. The capability `CAP_NET_ADMIN` is required for network administration, which can be revoked from processes to force a certain virtual network interface setup. Even when a function may be called as of the capabilities, its features may be limited by resource limits [EK10] (e.g. the non-POSIX limits `RLIMIT_RTPRIO` and `RLIMIT_RTTIME` related to real-time scheduling). Even more fine-grained access control for specific processes can be achieved via Linux Security Modules (LSM) [WCM+02] and Secure Computing (Seccomp) [Cor09]. Additionally, each process can be in several kinds of not standardized namespaces [WK17], which limit the visibility of kernel objects (e.g. other processes, network interfaces, mount points). A process can be further restricted by control groups (cgroups) [HK21]. For instance, the CPU time and block device access for all processes of a user can be limited to a specified quota via dedicated cgroups. These kernel features are used by and together with container technologies such as lxc [Can22] [Can22] and Docker [Mer14] to create isolated process spaces, in which processes have only a limited view of the system according to a container configuration and security profiles [Cha17; LMC+20; SCC+20]. Overall, Linux is a POSIX-based operating system with rich isolation features, which not only improves security, but which also provides architectural benefits including the possibility to use containers as self-contained provisioning units while enforcing compliance of a contained application to its declared interfaces. This is why we use Linux including containers for the runtime platform concepts described in this theses.

However, we are aiming at hard real-time systems. Approaches such as the PACORA framework [Bir14] (the thesis also contains an overview) have been proposed in order to guarantee QoS (in this case, deadlines) of a set of distributed soft real-time applications by optimizing resource allocation (compute, communication, capacity). Such approaches could be utilized in an architecture as proposed in this thesis, but within our scope, we do not try to minimize the allocated resources – instead we assume the allocated resources as demanded by a given model well cover the worst-case requirements so that deadlines are guaranteed in any case. Either way, we need some RTOS features for our real-time applications, too. Linux provides the RAM-only filesystem tmpfs [Ker16], which we can use (also within containers) to avoid disk I/O by time-critical processes. The Linux kernel supports following **scheduling** policies that can be requested by the processes via *sched_setscheduler()* or *sched_setattr()* (cf. [KZL+22]). Normal processes run with scheduling policy `SCHED_OTHER`, which under Linux is the completely fair scheduler (CFS). The only requirement by POSIX is that a priority can be set, which on Linux must be equal to 0. Thus, all normal processes have the same priority. The scheduler runs periodically by means of a configurable timer interrupt (e.g. every 10 ms) and when processes voluntarily release the CPU or become runnable. If no other process is runnable, the scheduler picks one of the runnable processes in the normal thread list. A nice value is used to fairly balance this selection per default, which can also be manipulated from user space to de/prioritize processes. Besides that, the Linux kernel supports real-time scheduling policies. The more basic policies also included (optionally) in POSIX are `SCHED_FIFO` and `SCHED_RR`. Such processes can have higher priorities up to 99. For FIFO processes, the scheduler selects from the highest priority thread list the longest-waiting runnable process and lets it run until it voluntarily gives up the CPU, reaches its runtime limit (`RLIMIT_RTTIME`), or a higher-priority task becomes runnable. When a FIFO process is preempted by a higher-priority

task it is enqueued at the head of its priority's thread list, otherwise at the tail. RR processes (round robin) may also be preempted and re-enqueued when a configurable time quantum exceeds (e.g. 100 ms). Linux also provides the non-standard real-time scheduling policy `SCHED_DEADLINE`, which enables isochronous task execution. Such processes automatically have the highest priority and will preempt any other process when becoming runnable. If multiple processes use this policy, they are scheduled via the earliest deadline first algorithm. When the process requests this policy, it passes a period, a runtime, and a deadline to the kernel. Normally, the process will become runnable at all periodic offsets from the request time. If the deadline is the closest, the process will be set to running and keeps running until it voluntarily gives up the CPU or until its runtime is used up. The isochronous instants may be shifted by the kernel if the process should not release the CPU by calling *sched_yield()* until the deadline, especially when it runs out of runtime. To achieve real-time behavior, additional real-time tuning such as core pinning and process memory locking is required in addition to using real-time scheduling policies. In particular, the `PREEMT_RT` patches can be used to reduce the scheduling latencies caused by non-preemptive parts of kernel-space functionality, which lead to priority inversion [RMF19]. For using real-time scheduling policies such as `SCHED_FIFO` within containers while isolating their scheduling to a cgroup quota like we do our concepts, the kernel must be configured for `RT_GROUP_SCHED` so we do not need to assign an `RLIMIT_RTTIME` per process. Combining the fully preemptive kernel and the real-time cgroup scheduling is not supported officially [RMF19], so that requires an additional kernel patch and caution in configuring the system.

Linux not only supports time-triggered tasks via `SCHED_DEADLINE`, but also via custom timers and also event-triggered tasks. The basic abstraction for this is provided by **signals** (either directly or via system functions). Kernel and application processes in the same process group can generate signals for specific processes or threads via their process or thread identifier (PID/TID). This can be done in case of timer expiration, hardware events, or by system and application events. The list of signal types is fixed. Some signals directly manipulate the process execution (e.g. terminate on `SIGKILL` or continue on `SIGCONT`). Other signals can be accepted and processed actively by the target process or thread (e.g. `SIGINT`). A process or thread may also provide a pointer to a handler function triggered by the OS, which will lead to preemption ("interruption") of the normal process execution. It depends on the signal whether the application can change the default action for a signal. Signals can be blocked by each process's signal mask, which keeps them pending until they are unblocked, or they can also be ignored, so they are discarded. Signals may or may not be queued and prioritized (prioritized, queued signals are called "real-time signals" in POSIX), and some may contain an application-defined integer or pointer value. Thus, the signal concepts provide features for process management and **interprocess communication**, which together with supported scheduling policies and the `PREEMTP_RT` patches can make Linux suitable as an RTOS. This requires mechanisms for maintaining inter-thread consistency (semaphore, mutex, spinlock,...) while preventing priority inversion (priority ceiling and inheritance). The priority ceiling of a mutex can be set dynamically by calling *pthread_mutexattr_setprioceiling()*. This can be used to avoid priority inversion by setting it to the highest priority of any thread that might lock it, because threads will temporarily inherit this priority. However, for other resources such as semaphores, the POSIX standard requires applications to implement priority inheritance on their own via corresponding calls to *pthread_setschedprio()*. Additional care is required because POSIX defines several **option groups** (defining optional OS features), which are important for

real-time systems, and the support of which depends on the Linux kernel configuration. The Realtime Option Group includes the options to prioritize processes, to lock memory in the RAM, and to use shared memory and POSIX message queues for high-performance inter-process communication (IPC). It also includes the options to synchronously force pending file I/O operations (in addition to asynchronous I/O with a completion signal), or to prioritize them in the I/O queue according to the calling process's priority. The Advanced Realtime Option Group additionally provides an implicit CPU-time clock for each process counting the runtime, monotonic clocks which do not jump backwards, the possibility to spawn processes with different scheduling parameters (versus forking, which normally inherits them), a sporadic server scheduling policy, for running asynchronous events in a process with constantly reserved CPU bandwidth (not directly supported under Linux, however available as `SCHED_DEADLINE`), and typed memory objects for name-based memory mapping of I/O resources. The Realtime Threads and Advanced Realtime Threads Option Groups provide real-time scheduling features on thread-granularity, for example setting thread priorities. Finally, due to the high customizability and complexity of Linux, at least safety-critical hard real-time systems do not seem possible today with some of the richer features enabled (e.g. cgroups), as latencies and worst-case execution times (WCETs) are not predictable, though there are some advances [RMF19]. These aspects should be kept in mind when using a POSIX-based OS or Linux in particular as an RTOS.

Besides RTOS and container features, we also use **advanced networking** features of Linux [HGM+22] for orchestrating the containers and shaping the traffic. Figure 2.7 shows a simplified overview of the packet flow through the kernel's network stack. Per default, each network interface – including virtual ones – has two FIFO queues attached. When the driver receives a new packet (ingress) an *sk_buff* structure is allocated at the socket buffer and a pointer is passed to the ingress Qdisc (enqueue). The kernel asks the ingress Qdisc "occasionally" for the next incoming packet (peek and dequeue) and gets the oldest one, per default. Firewall rules created with *iptables* [EBJ+22] are evaluated at the green steps in the flow. The pointer to the packet is first passed through the prerouting chain, which might for instance modify the destination, before a routing decision is made based on the routing table. If the destination is an IP address of a visible device of the local machine the packet moves up the protocol layers to the application process. When a packet is sent by an application, an initial routing decision is made to decide over which network interface the packet is sent. A pointer to the socket buffer is passed through the corresponding output chain, where firewall rules may again modify the destination, for instance. Thus, the routing decision is checked again after the output chain. Finally, the pointer is enqueued to the egress Qdisc of the network interface. The kernel "occasionally" asks the Qdisc for the next packet that should be transmitted. Note that in case of containers, where we usually use virtual pair devices – one of them called *eth0* in the container namespace and the other in the host namespace – a packet leaving the egress Qdisc on the one end is enqueued to the pair device on the other end in both directions (packets to and from the container). In earlier versions of our platform prototype, the dynamic orchestration of application containers was done via firewall rules based on *iptables*. Unfortunately, when we evaluated this prototype we measured that modifying the firewall rules took between 40-57 ms [TK19a]. To enable dynamic reconfigurations without downtime this must be done during a cycle turnover, though. As we wanted to enable periods of 100 ms and below, that was unacceptable. Therefore, we redesigned the dynamic orchestration and moved this Kernel-space functionality to a custom *barrier* queueing discipline (Qdisc) that we

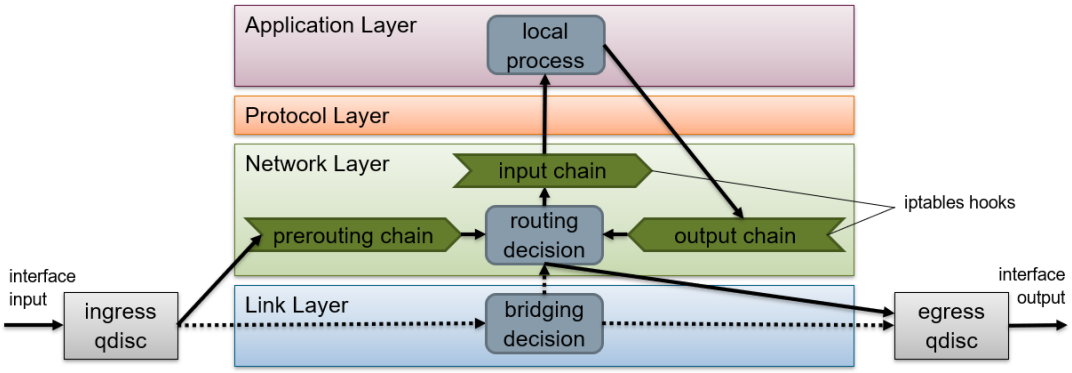

FIGURE 2.7: Location of queueing disciplines (Qdisc), the firewall, and the routing decision in the packet flow through a network interface in Netfilter based on the much more detailed (and accurate) original by Engelhardt [Jan19].

needed anyways as described in chapter 5. Note that a Qdisc does not necessarily need to return a pointer during the peek operation, whereas firewall rules must pass or drop a packet. For instance, the token bucket filter Qdisc does not yield a packet while configured rates are depleted. Additionally, the ordering and headers (even contents) of packets may be manipulated, and packets may be even deleted by a Qdisc at any time. With adequate network configurations it is possible to prioritize packets by their protocol or by their destination, for instance, and even to send them in specific time slots as in TSN [ZHLL18]. In fact even the simple FIFO Qdisc in Linux prioritizes packets by the Type of Service field of IP packets. Thus, the generic Qdisc concept in the network stack of Linux enables a wide range of traffic shaping approaches. A special-purpose kind of traffic shaping and scheduling (to achieve LET-compliant UDP communication) is also provided by our *barrier* Qdisc besides dynamic orchestration.

2.3 Formal Methods

After elaborating on related architectural principles and technologies for reconfigurable distributed control systems, this section describes more formal foundations and related work. The basic concept for mathematically describing and analyzing computer programs is to define a notation and semantics that result in some kind of transition system (or state machine, automata) on which a calculus is defined. A program run is thus a sequence of states and output information, which is determined by the program rules and input information. This is usually done by algebraic means, i.e., by defining the sets of program terms and symbols that can be used to model programs, and then using variable assignments to represent a dynamic state and a strict semantics of what each program step (as of the terms) does to the state of the program, including the program terms. This approach was also called evolving algebras in general [Gur95], as many specialized formal methods were proposed. Formal methods are important tools especially in safety-critical systems, or in general, when correctness of a program in any circumstances and corner cases is of highest importance. We first describe formal methods used for specification and analysis of distributed control systems in section 2.3.1. Our component and reconfiguration model in chapters 3 and 4 is specified using some flavors of Maude [CDE+16], a framework for term rewriting logic, which we describe in section 2.3.2. Finally, we give an overview of formal frameworks for dynamic reconfiguration in section 2.3.3.

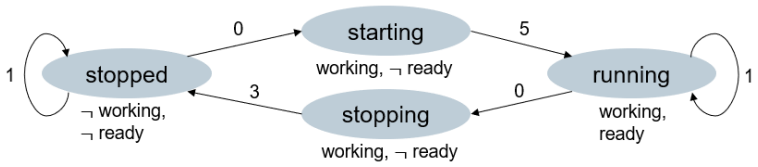

FIGURE 2.8: A timed Kripke structure of a system which needs a fixed time to start and stop and can stay stopped and running for arbitrary time. From any state the stopped state is reachable within 8 units of time, so that the atomic proposition *working* turns *false*, i.e., the timed CTL formula $EG_{\leq 8}\neg working$ is always satisfied.

2.3.1 Timed Evolving Algebras

In context of real-time systems we don't only want to model and analyze whether certain states are reachable at all, but also in what time. Thus, the transition system must somehow include passage of time, which increases the complexity (due to more states and/or transitions). Various approaches for this were proposed based on different untimed models. The naive approach is to replicate the states and transitions to cover all possible values of the clock, which is obviously computationally complex. Timed Petri nets [Ram73] add a fixed duration to each transition in a Petri net and record the time of each transition firing in a firing schedule. The more general time Petri nets [MF76] (recently standardized in IEC 15909 [ISO19]) instead use a delay interval with minimum and maximum time between enabling a transition (as of the conditions) and the actual firing of the transition, which takes no time, but can be regarded as the end of the transition duration. To handle the complexity resulting from the variable durations during reachability analysis, state classes can be used [BD91]. Timed automata [AD94] were the first to use a dense time base (real-valued clocks). This has been enabled in timed automata by constructing a reduced region automaton from equivalent clock regions to cope with the infinite (time-extended) state space induced by the dense time base [AD94]. More abstractly, partial-order reduction can be used to cope with state space explosion by only analyzing a graph of independence-preserving regions instead of the full transition system [ABH+97]. As such optimizations and model checking based on timed CTL [ACD90] are supported by tools such as UPPAAL [BDL04], timed automata are widely used to specify and analyze real-time systems.

There are similar timed concepts based on Kripke structures (see [LÁÖ15] for an overview) that allow for model checking with timed CTL, too, and are less complex to evaluate than timed automata [LMS02]. Timed Kripke structures [LÁÖ15] support both discrete (pointwise) and dense (continuous) time and they can be used with Real-Time Maude [ÖM07] (see section 2.3.2). As this is a formal foundation of this thesis, we describe timed Kripke structures more in detail based on [LÁÖ15].

Definition 1. *A timed Kripke structure is a tuple* $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$ *where*

- S *is a set of states,*
- $\mathcal{T}$ *is a time domain (e.g.* $\mathbb{N}$, $\mathbb{Q}_0^+$*),*
- $\longrightarrow \subseteq S \times \mathcal{T} \times S$ *is a total transition relation with duration, and*
- $L : S \rightarrow \mathcal{P}(AP)$ *is a labeling function over a set of atomic propositions* AP*.*

Figure 2.8 shows an exemplary timed Kripke structure in the typical graphical notation. The basic idea is that while being in state $s \in S$, the propositions $L(s)$ hold. Transitioning from state s to s' via transition $s \xrightarrow{t} s' \in \longrightarrow$ takes t time units. In

the pointwise interpretation, the state atomically changes to s' exactly t units of time after entering s, if no other transition is taken. In the continuous interpretation, intermediate instants can be visited in addition, while still staying at s. For modeling this, a path over $\mathcal{TK}$ is defined as an infinite sequence of configurations $\langle s, \delta \rangle \in S \times \mathcal{T}$, with δ being the time since entering state s. A timed path over timed configurations $\langle s, \delta \rangle @c \in S \times \mathcal{T} \times \mathcal{T}$ additionally maintains a global system time c, which always starts at 0. Thus, (timed) paths in the pointwise interpretation only reach configurations $\langle s, 0 \rangle @c$, with c increasing by the duration of the taken transitions. A continuous interpretation would in general allow the same paths, but also paths with an infinite amount of additional intermediate configurations $\langle s, \delta \rangle @(c + \delta)$, as long as there is a transition $s \xrightarrow{t} s'$ with $\delta < t$. Nevertheless, explicit-state model checking of continuous interpretations with timed CTL is possible by analyzing a transformed timed Kripke structure with pointwise interpretation. For this, each tick transition $s \xrightarrow{t} s'$ is split into multiple smaller tick transitions over intermediate states equivalent to s. The durations of these intermediate tick transitions is set to half of the greatest common divisor of all transition durations and interval bounds in the timed CTL formula. Similar to the region graph in timed automata this ensures that all relevant points in time are visited by a discrete interpretation. This way, Kripke structures are efficiently enriched which quantitative time modeling and analysis. We refer to S# ("safety sharp") [HLR16], a framework for simulation and analysis of safety-critical systems that uses (untimed) Kripke structures. In this framework, C#-based controller components and similar components modeling the environment behavior perform lockstep, blackbox cycles (macro steps) under ZET abstraction. Only the ground state between macro steps is analyzed, by translating it to a Kripke structure. Additionally, the ground state can be manipulated, which is especially used for fault injection. This way, an automated Deductive Cause Consequence Analysis (DCCA) can be done to identify fault sets leading to hazards based on the behavior of the real controller code in the simulation loop. At the end of section 3.3 we mention an approach to model check systems in our formal framework, which in the end is based on timed Kripke structures. It may be an interesting future direction to explore applicability of the S# concepts to our model and architecture.

We finally mention other approaches called process algebra, which focus on composition mechanisms with messaging for modeling distributed systems. One prominent example is communicating sequential processes (CSP) by Hoare [Hoa78], for which timed CSP [DS95] adds time in a few additional or modified operators. In timed CSP a process can wait until a certain time has passed (instead of instantly skipping), it can timeout when waiting for a message, and it can be interrupted at a certain time. Their composition mechanisms have been adopted to timed automata, while their semantics – especially scheduling assumptions – lead to difficulties in analyzing systems [OD08]. In case of timed CSP it was complained that a constant reaction time between processing of consecutive messages is assumed [Old98]. However, these formal methods in raw form are not well suited for modeling and analyzing dynamic systems with objects being created or removed as in dynamic reconfigurations. The following section describes an approach which is better for modeling such aspects and for which formal analysis based on timed evolving algebra concepts is supported.

2.3.2 Term Rewriting

A good and maybe the only method to specify and analyze dynamic reconfigurations is (term) rewriting logic [Mes92], which we already find in Hammer [Ham09]. It allows to describe a system and its reconfiguration in an object-oriented way and thus understandably, but still precisely. We specify our component model and runtime platform concepts in Full Maude [CDE+16] (see Chapters 3 and 4). We also use the Real-Time Maude extensions [ÖM07] for modeling passage of time during computation and communication. With the underlying concepts supported by Real-Time Maude [ÖM02], rewriting logic can be used to specify real-time systems according to the common models described previously in section 2.3.1 (e.g. timed automata). Unlike those formalisms, Real-Time Maude allows for dynamic object creation and deletion (among other benefits), which makes it natural to model object-based distributed real-time systems [Ölv14], and especially to model component-based distributed real-time systems and their dynamic reconfigurations. We give a brief overview of term rewriting with Real-Time Maude.

```
op nil : -> List [ctor] .
op _._ : Qid List -> List [ctor] .
op length : List -> Nat .
vars I J : Qid .
var L : List .
eq length(nil) = 0 .
eq length(I . L) = 1 + length(L) .
rl [collapse] : I . J . L => append(I, J) . L .
```

LISTING 2.1: An algebraic list specification and a rewrite rule in (Core) Maude.

Listing 2.1 contains a functional specification of a list of quoted identifiers in (Core) Maude based on an example from the book All About Maude [CDE+07]. Three sets (*sorts*) are used as types: *List*, *Qid*, and *Nat*. *Qid* (quoted identifiers, e.g. `'id1`) and *Nat* (natural numbers) are predefined. The elements (members) of *List* are given by two constructor operators (indicated to Maude by the trailing *ctor* operator attribute). `op nil` has no attributes and is thus a constant – the empty list. `op _._` is a binary inline operator with the dot as the operator symbol and the underscores to indicate the non-default attribute positions. It takes a *Qid* and a *List* and yields a new *List* – with the *Qid* at the head of the given *List*. Thus, the set of lists contains exactly *nil* and all possible chains of applications of the dot operator with *Qid*s and *nil* at the tail. `op length` is a unary operator which takes a *List* and yields a *Nat* – the number of elements in the list. This is defined by two equations using variables, pattern matching and recursion. A term is either a constant, a variable, or the application of an operator to a list of argument terms [CDE+07]. Before rewriting, Maude first tries to simplify and reduce terms by applying defined equations, replacing lefthand terms with righthand terms. For example, `length('a . 'b . 'c . nil)` can be reduced to `1 + 1 + 1 + 0` by applying the length equation three times, which can be reduced to the third successor of zero (or just 3, as arithmetic operations and number symbols are supported for *Nat*). Afterwards, Maude searches for applicable rewrite rules. Listing 2.1 also contains an example rewrite rule labeled *collapse*, which matches all lists with at least two elements. When applying the rule to a term, Maude replaces the matching term or subterm with the righthand term of the same kind, which is called a rewrite. The *collapse* rule replaces a list term with a modified list term (the types are inferred from the operators), in which the head is a new *Qid* calculated from the *append* operator applied to the two head elements. For example, `'a . 'b . 'c . nil` could be rewritten to `'ab . 'c . nil` in one step and then to

`'abc . nil`, after which no further progress is possible. This can be done using the *rewrite* command like this in the Maude command line: `rewrite 'a . 'b . 'c . nil`. If multiple rules match, then Maude uses a rewrite strategy to select one of them, for example, fairly balancing between them. Thus, the rewrite command yields only one possible trace from the initial term. Using the *search* command, we could check that the length of all final lists resulting from a given input will be equal to one (omitting the module argument for brevity):

```
search : 'a . 'b . 'c . nil =>! L:List such that length(L) =/= 1 .
```

Maude will then search all reachable end terms (due to the `=>!` operator) from the given input and print those which do not have the length of one. Depending on the state space spanned by applicable rewrite rules this can take very long. When considering all possible rewrite sequences from all terms, a transition system is implicitly defined, with the reduced terms as states and the rewrite rules as transitions. It is also possible to evaluate LTL formulas on rewriting specifications by mapping the possible markings to a Kripke structure [CDE+07].

Full Maude provides syntactic sugar to Core Maude for better specifying object-based distributed systems with objects and messages between them. Object-oriented modules implicitly include the `CONFIGURATION` specification, parts of which are shown in listing 2.2. Elements of the sort *Configuration* are used as root term for rewrites

```
op __ : Configuration Configuration -> Configuration [ctor config assoc comm id: none] .
op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
```

LISTING 2.2: Configurations and Objects in Full Maude.

of Full Maude specifications. Two configurations separated with whitespace yield another configuration. Two subsorts *Object* and *Message* exist besides the *none* constant. Thus, a configuration is an unordered (due to the associative and commutative equivalence defined by the attributes `assoc` and `comm`) set of objects and messages. As specified in the other `op` term, an object is a 3-tuple of an object identifier (typically *Qid*), a class identifier as of a `class` declaration, and an attribute list. Attributes are key-value pairs with key identifiers and value sorts as declared in the corresponding `class` declaration term. Listing 2.3.2 shows a Full Maude specification of a bank account (cf. [CDE+07]).

```
class Account | bal : Int .
msgs credit debit : Oid Nat -> Msg .

vars A B : Oid .
vars M N N' : Nat .

rl [credit] : credit(A, M) < A:Account | bal : N > => < A:Account | bal : N + M > .
crl [debit] : debit(A, M) < A:Account | bal : N > => < A:Account | bal : N - M > if N >= M .
```

LISTING 2.3: Specification of a bank account in Full Maude (cf. [CDE+07]).

A valid example configuration according to this specification would be:

```
credit(A-002, 100) < A-002 : Account | bal : 1000 >
```

The object *A-002* is an instance of the *Account* class, and the *credit* operator application is a message according to the special message operator declaration with `msgs`. Using the rewrite rule *credit* this configuration could be rewritten to

```
< A-002 : Account | bal : 1100 >
```

Full Maude will reorder objects and messages to match a rule and try to balance the processing of different message types (due to the attributes `config` and `object` in listing 2.2), if multiple messages can be processed by matching rewrite rules.

This way we can naturally specify object-based and distributed systems (though a more detailed messaging specification is needed in our case). The *rewrite* and *search* commands as well as LTL-based model checking features are also applicable to Full Maude specifications. For example, we could search whether we can reach a state in which an account gets a negative balance using this command:

```
search debit(A-002, 500) debit(A-002, 600) < A-002 : Account | bal : 1000 >
  =>* < A-002 : Account | bal : N:Int > C:Configuration such that N < 0 .
```

As the rewrite rule *debit* has a condition which ensures that it is only applied if the balance is sufficient, the search will yield no solution for a negative balance.

Real-Time Maude extends Full Maude for specification and analysis of the quantitative temporal behavior of systems. For representing time a sort *Time* is introduced, for which different time domains are predefined based on natural numbers (discrete) and rational numbers (dense). A time domain must have a zero element and support the operators $+, \dot{-}, \leq, <$ as known from natural and rational numbers [ÖM02]. Based on this, timed rewrites can work on tuples *Configuration* $\times$ *Time* instead of only configurations, with an additional monotonous system time. The previously described ordinary rewrite rules from Full Maude become instantaneous rewrite rules, as they stay as-is and will work on the *Configuration* part, only (i.e., no progression of time). Additional tick rules must be defined, which allow progression of time and beyond that only change the *Configuration* to reflect the effects of the passed time (e.g. counting down a timer). Real-Time Maude adds syntactic sugar for this and provides guidelines in defining real-time systems. The tick rule for object-oriented specifications is usually defined as in listing 2.4 [Ölv14].

```
vars T T1 T2 : Time . vars C1 C2 : Configuration . vars M : Msg . var SYSTEM : Configuration .

op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(none) = INF . --- infinity value
ceq mte(C1 C2) = min(mte(C1), mte(C2)) if C1 =/= none and C2 =/= none .

op timeEffect : Configuration Time -> Configuration [frozen (1)] .
eq timeEffect(none, T) = none .
ceq timeEffect(C1 C2, T) = timeEffect(C1, T) timeEffect(C2, T) if C1 =/= none and C2 =/= none .

crl [tick] : {SYSTEM} => {timeEffect(SYSTEM, T)} in time T if T <= mte(SYSTEM) .
```

LISTING 2.4: The usual tick rule with operators *mte* and *timeEffect* [Ölv14].

The conditional rule allows progression of a variable amount of time (T) as long as T is smaller than or equal to a maximum allowed time elapse, which as calculated from the system configuration via the *mte* operator. The system configuration is updated accordingly by the *timeEffect* operator. How Maude will determine a value for T is specified by the selected time sampling strategy. Per default, a user-given tick value is used if it is smaller than the *mte* value. In the maximal strategy the *mte* value will be used unless it is infinity, in which case the user-given tick value is used. The curly brackets are introduced so that the tick rule will match and update the whole system term instead of only subterms of the configuration. Both *mte* and the *timeEffect* operators are defined by system-specific equations for the corresponding object and message types. For instance we could model a periodic behavior by a clock attribute of one of the objects, which we count down as time progresses and reset after instantaneously doing the periodic work:

```
class Periodic | clock : Time, period : Time .
eq mte(< P : Periodic | clock : T >) = T .
eq timeEffect (< P : Periodic | clock : T >, T2) = < P : Periodic | clock : T monus T2 > .
rl [reset] : < P : Periodic | clock : 0, Period : T > => < P : Periodic | clock : T > .
```

The *mte* function returns the clock value, while the *timeEffect* operator decreases the clock value accordingly. An instantaneous *reset* rule resets the clock value when it has reached zero so that progression of time is possible, again. For timed rewriting we can use the *trew* command from Real-Time Maude:

```
trew {< P : Periodic | clock : 100, Period : 100 >} in time <= 200 .
```

This will apply matching instantaneous rules and tick rules until a total of 200 units of time have passed, which can only happen in the tick rule. In this case a possible trace could be *tick(100)*, *reset*, *tick(100)*, leading to the final term

```
{< P : Periodic | clock : 0, Period : 100 >} in time 200 .
```

Using a corresponding *tsearch* command it is also possible to check whether certain configurations are reachable from an initial state in bounded time. LTL formulas can be evaluated like in Full Maude if a time limit is given, and Real-Time Maude also features a timed CTL model checker based on timed Kripke structures [LÁÖ15], so that absolute properties of the implied transition system can be proven with respect to quantities of time. Section 3.3 includes an example showing how reactivity and reachability of hazards can be analyzed through model checking. However, the state space of our model is very large, so model checking features were used rarely in the scope of this thesis. Still we used Real-Time Maude for platform-independent specification and analysis of our runtime platform concepts for modular and evolvable distributed real-time systems. Though there is still a gap between our Maude-based model and the Real-Time Container Architecture prototype, the model is very close to the implementation, especially regarding reconfigurations. In fact, the Maude model existed before we started with the implementation of the prototype, as the platform-independent specification helped to abstract from the distracting realities of Linux. The similarity to the implementation is important for trusting that the model "faithfully reflects reality" [Gur94]. One of the main goals of the specification is to formally describe and analyze reconfigurations, which we address by using a similar reconfiguration plan description language in the model and the implementation. For instance, by simulating a concrete reconfiguration plan in Real-Time Maude, we can analyze its consistency and effects before executing it in the real system. Other analysis aspects include the reactivity of a system, also during reconfigurations. Sections 3.3 and 4.2 describe more in detail how to use Maude for these analysis purposes in context of our formal framework.

2.3.3 Component Frameworks for Dynamic Reconfiguration

We give an overview of existing formal frameworks for dynamic reconfiguration. Though we are aiming at dynamic reconfiguration of distributed control systems, we include approaches for non-real-time systems, and non-distributed systems, and also non-component-based systems. We focus on reconfiguration specification and analysis concepts for the software level, not the technical process level, even though the impact of the software reconfiguration on the technical process should be analyzable, too.

We first describe component frameworks for **dynamically reconfigurable distributed systems**, which do not support real-time systems. The first concept for dynamic updates was published in 1976 by Fabry [Fab76]. A module (a function) is replaced during system execution by a new version as follows: The new version is deployed in addition, and an indirection mechanism is used by the callers, which keeps pointing to the old version and switches to the new version once the old one is not

used anymore. The indirection mechanism may use locking so that any internal initialization work (including transformation of data structures) is done during the first invocation, only. The first approach for distributed module updates was proposed by Bloom for the Argus framework [Blo83]. The approach supports "multi-site replacements" by means of a replacement environment for the user, which provides replacement and state transformation commands that can be executed remotely on the different nodes from the user side. The user program that uses these commands to achieve a distributed replacement can be seen as a reconfiguration plan. Remote procedure calls between the different modules may be preempted and aborted during the concurrent reconfiguration. Alternatively, lock-based atomicity was proposed, i.e., requiring exclusive access to the module by all users, including reconfiguration operations. The consistency of replacements was analyzed with a dedicated formal model. It models a module as a state machine which takes input events (invocation messages) and produces output events (reply messages), maintaining a history abstraction within its state. An implementation of a module interface (called abstraction) may only generate a subset of the event sequences (futures) declared by the abstraction. In essence, the main consistency criterion is that the two modules need to be replacement compatible in the sense that the replacement should not yield different futures in a sequence of invocations, so that all user transactions stay consistent regardless of when the change happens. Kramer and Magee proposed a quiescence-based approach for distributed systems [KM90]. Quiescence is the property that a node is "consistent and frozen", i.e., it is not working and will not be working on any transactions (a bi-directional exchange of messages between two nodes initiated by one node). Before replacing a node, it is brought to a quiescent state by transitively blocking all nodes which may directly or indirectly trigger a transaction within that node and then the node itself. For this, an acyclic precedence graph of transaction invocation relationships between the nodes is needed, so that the blocking can be ordered accordingly. Circular dependencies can be resolved when consequential transactions do not lead to further transactions. In this case, replies by nodes requested to be blocked are still allowed in order to reach a quiescent state. Only after blocking succeeded, the replacement is performed, so that it can not disrupt any ongoing or incoming transactions. The effects of the reconfiguration were analyzed by means of predicate logic. Tewksbury et al. [TMM01] extended this concept for live upgrades of CORBA [OMG12] applications with distributed component replicas. Their framework first enables replication of components for fault tolerance by means of interceptors within the CORBA platform, so that each replica processes each request and duplicate responses are eliminated. By analyzing the CORBA method invocation graph at design-time, nested invocations and thus the ordering of replacements is derived, so that quiescent states can be reached consistently as described before. In a first phase, all components and replicas are replaced transparently with a generated, backwards-compatible intermediate version that is able to provide both the old and the new implementation and interface. After all intermediate versions are ready, an atomic switchover of all participating components is requested, so afterwards only the new component versions are active. Finally, each component is replaced, again, so it only contains the new version. Each of the replacements requires quiescence (i.e., blocking) as well as state transferal and state conversion (at least copying of the buffered requests). Thus, the replacements happens backwards along the invocation graph, while new forward transactions are buffered, with replicas being replaced one after another within that order. The different evolution steps – especially the atomic switchover – must happen at the same point in the transaction history regardless of buffered requests. Therefore, their

concept is completed by ensuring totally-ordered multicast of messages within the middleware. A formal approach to such reconfigurations (with transferal of internal data and buffered requests) was described by Hammer [Ham09]. An algebraic specification of a request/reply-based component model and a reconfiguration algorithm is provided based on term rewrite logic with Maude. This modeling approach inspired us to use Real-Time Maude in this thesis. A reconfiguration plan is modeled as a tuple $\Delta = (A, R, \alpha, \rho, \delta, \varsigma)$, with new components A, components to remove R, connections α of new components, rewirings ρ of remaining components, and message retainments δ and state transfer ς from old to new components. Those are all sets and relations, and lead to following reconfiguration procedure via corresponding reconfiguration transition rules, which are applied in addition to the normal component execution and communication rules. First block all components to be removed as of R by changing their state (they must eventually stop). Then instantiate all new components in A and wire their required services according to α. Also rewire the remaining components according to ρ before changing the state of new components to *initialized*. Afterwards, the states and then the messages of components in R are copied to the new components in A according to ς and δ. Finally, old components are removed and new components are activated. This concept ensures consistency of the reconfiguration, also in the distributed case, with downtime between stopping old components and starting the new ones. Hammer also gives a comprehensive overview of similar work on reconfigurable "Java-like" (in our terms) component models. This includes a classification whether and how the different approaches address state transfer (messages, user-level, ...) and atomicity (e.g. quiescence, tranquility, ...), and how reconfigurations are triggered (external, programmed, ...) and described (reconfiguration plan, discovery, reconfiguration services, ...). For more work into this direction without hard real-time we thus refer to his work.

Many of the aforementioned concepts have a formal backing based on some model similar to communicating sequential processes (CSP [Hoa78]). However, **distributed real-time systems** usually do not communicate via arbitrarily blockable request/reply communication, because they must have a time-deterministic communication and processing behavior from sensors to actuators. Therefore, a reconfiguration concept for such a system must provide a solution for avoiding, coping with, or at least analyzing temporal impact, including any distributed coordination of the reconfiguration. The first approach for backwards-compatibly updating individual components in real-time systems was proposed in 1996 by Sha et al. [SRG96]. After creating the new component, it is fed with real input data until it is ready, while its outputs are only monitored. As soon as the outputs of the old and the new component versions converge, their roles are switched, i.e., the old version is turned off and the new one is unmuted. If multiple component should be replaced at the same time, only the switch must be done simultaneously (which, however, can be difficult, as later chapters will show). This concept can be seen as the basis for the existing approaches to updating real-time systems (cf. seamless upgrade [FKST23]). In section 2.2 we already mentioned some approaches in automation, avionics, and automotive systems. We mentioned our approaches in the Real-Time Container Architecture, the formal specification and analysis of which is described in chapters 3 and 4 based on term rewriting logic. There is no comparable approach to our Evolving Dataflow Graph for analyzing dynamic reconfigurations of distributed LET-style systems. Hamann et. al. [HDK+17] proposed a formal analysis of the end-to-end behavior for system-level LET in AUTOSAR classic without dynamic reconfiguration. In AUTOSAR adaptive, the reactor model was proposed

in DEAR [MGLC20], for which a safe-to-process analysis was proposed to increase the reaction chain consistency. Using the scheduling parameters from this analysis, an end-to-end reaction time analysis could be performed which considers the complete processing chains [MGLC20], but it has not been described more in detail to our knowledge. Dynamic reconfiguration is not included in the analysis. As also mentioned, a formally backed concept for model-preserving runtime patches [KLMS11] was proposed for HTL [HKMS09]. The concept can atomically add, remove, and replace individual software components by adding operational semantics rules for a runtime patcher (the design of which is not further specified). It is also discussed that multiple components can be replaced sequentially (or "decomposed"), if the components are independent, or synchronously in concurrent quiescent states. A runtime patcher component observes all components' states and recognizes which of them are quiescent. If a component should be replaced, this is only done in quiescent states, which are also a valid initial state for the replacement. The replacement transition includes termination of the old component (if not strictly adding) and initialization of the new component (if not strictly removing). Though HTL is a language for distributed real-time systems, the concept does not discuss how to synchronously achieve this in the distributed case. State transfer and communication reconfiguration are not considered, either, so no structure-changing reconfigurations are possible. As no structural changes are caused, no end-to-end timing analysis is required for the concept. These aspects are considered by our concepts, so that multiple components can be replaced, or added and removed, including distributed breaking changes, that must be considered in the reconfiguration transition. In the context of PLC-like systems we mentioned a non-formalized concept for updating individual cyclic components [WRO09]. It performs the full replacement within the first regular execution cycle, including termination, initialization and state transfer. We also mentioned a similar, more formalized approach for live updates on PLCs in Cetratus [MM21]. An atomic switch from a primary and a secondary container is performed after in-field monititoring of the new logic. The full PLC program – a Petri-net – is replaced in this concept, including a transfer of the state with a transformation, if needed. The Petri-net is not distributed, not even over multiple containers on the same machine. In section 2.2.2 we described the architecture and update approaches for IEC 61499 [IEC12]-based distributed control systems, for instance the approaches by Zoitl et. al. [ZLMV10] and by Prenzel and Provost [PP17]. Dubinin and Vyatkin proposed a more unambiguous mathematical model defining the resulting transition system [DV07]. Drozdov et. al. provide an overview of formal approaches to IEC 61499 [DDPV21], including the use of timed automata and abstract state machines. Sünder et al. [SVZ13] described a formal framework for modeling and verification of IEC 61499 reconfigurations. They use the term downtimeless system evolution (DSE) to emphasize that a continuously running control system is gradually changed both regarding software and hardware. The reconfiguration analysis focusses on the time-critical reconfiguration sequence that changes the behavior of the control application. For this, an Net Condition/Event Systems (NCES) model [RH95] both of the control application and of the reconfiguration control application must be created. This describes the behavior of the FBs and of their interconnections as Petri nets, including the behavior and effect of reconfiguration control FBs. This model enables model checking to analyze whether the control application fulfills the requirements of consistency and timeliness, for instance.

To sum up, there are approaches to distributed real-time systems, which do not

consider dynamic reconfiguration. Other approaches allow updates to real-time systems without considering distributed changes. We see some approaches towards reconfiguration of potentially distributed real-time systems based on IEC 61499. However, our concepts avoid coordination communication in time-critical phases by exploiting the time determinism of the distributed LET-style application. Thus, we can completely avoid downtime in some distributed cases and minimize downtime in other cases, depending on whether time-consuming state transfer operations are needed. We keep the system's reactivity deterministic during reconfigurations. We are the first to enable compatibility-breaking distributed dynamic reconfigurations of real-time systems deterministically and with no quality degradation. As quality degradation cannot be avoided in all cases, we provide means to keep it deterministic and short.

Chapter 3

A Component Model for Modular and Evolvable Distributed Embedded Applications

This chapter specifies a component-based model for distributed real-time systems using Real-Time Maude [ÖM07][1]. The model was described less formally in earlier stages [TK17; TSK18; TK19a; TK19b]. It is designed to simplify integration of software components in distributed control systems regarding the aspects of composition, allocation and reconfiguration. Regardless of simplicity and evolvability, these systems must fulfill real-time requirements to achieve their purpose in the technical process. Especially the reaction time of the system is important, i.e., the time between observation and reaction of the distributed control system must be limited. The formal specification presented in this section thus describes the structures and procedures needed to enable three goals at once: Simplify integration, meet real-time requirements and enable dynamic reconfiguration. The proposed component model uses the logical execution time paradigm to achieve deterministic temporal behavior while avoiding priorities and time slices, because these lead to a big integration effort. In addition, it provides a structure to the real-time systems, which makes them reconfigurable during full operation (as described in chapter 4) while separating the concerns of functionality and reconfiguration. To achieve these goals, not only the software part needs to be modeled, but also the physical aspects of the system, i.e., of the hardware components and the network. Additionally, we have to use a minimal component model rather than a rich one, so that we can achieve an initial break-through over all phases, including dynamic reconfiguration.

This specification consists of a design-time model and a runtime model. The design-time model must be instantiated by engineers in independent engineering steps to define executable distributed real-time systems:

- Interfaces and functionality of software components
- Distributed embedded applications which consist of such components
- Microcontroller Units (MCUs) and the network topology of distributed systems
- Deployments of distributed embedded applications to distributed systems

The runtime model specifies the structure of the runtime environment and how it executes distributed embedded applications:

- Application containers which execute software components in an isolated way

[1] Maude code is "compactified" in this thesis: Only the most important pieces are included (e.g. omitting helper functions) and the strict syntax is relaxed using well-known elements from popular languages such as json and html to reduce the optical noise.

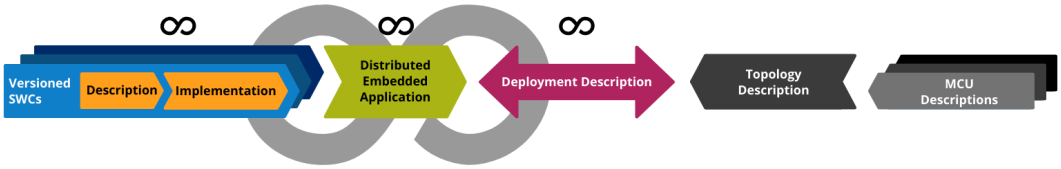

FIGURE 3.1: A schematic overview of the engineering steps (each of which provides a model fragment): Software components are described and implemented independently and wired in distributed embedded applications. They are deployed to a distributed system after describing the MCUs and the network topology.

- The system layer consisting of processing units, network interfaces and drivers
- The system execution management layer managing the nodes and containers

This is a platform-independent model, i.e., a technology-independent description of the concept. However, it provides a 'complete' abstraction of hardware and topology in the sense that besides the deployment mapping (and minimal platform-specific refinements of some steps) no glue code is required to manage and execute compliant distributed embedded applications on real hardware. The model can also be used to analyze the consistency and feasibility of specified distributed embedded applications. This is described in a third section in this chapter after specifying the design-time model and the runtime model. To illustrate the formal model and to explain the intention behind the different model elements, the *onBtnSwitch* example system (see section 1.2) is used as running example. The reconfiguration extensions are described separately in the subsequent chapter 4.

3.1 Design-Time Model

The following meta model can be used to define reconfigurable distributed real-time systems. Each section describes a model fragment, which also corresponds to an engineering step (i.e., instantiating the model fragment). The different model fragments are specified in the order of the engineering methodology proposed in [TSK18] (see figure 3.1). The major goal of the model is to decouple the engineering steps, so that they can be worked on independently: The model simplifies integration and reuse of software components in different distributed embedded applications. Additionally, distributed embedded applications are defined independently from a concrete allocation to nodes. Finally, the model builds the basis to enable modifications of distributed embedded applications (e.g. updating software components) during operation using reconfiguration plans. Thus, the engineering steps are not ordered strictly – especially, as they can be done iteratively.

3.1.1 Software Components

According to Szyperski [Szy98] *a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* Hence, in our component model we have to make explicit the functional and non-functional interface and dependencies to a component's execution context. This information is needed for integration both during design-time by engineers and engineering systems and for orchestration and execution during runtime. Additionally, the component model must define the base operational behavior of software components,

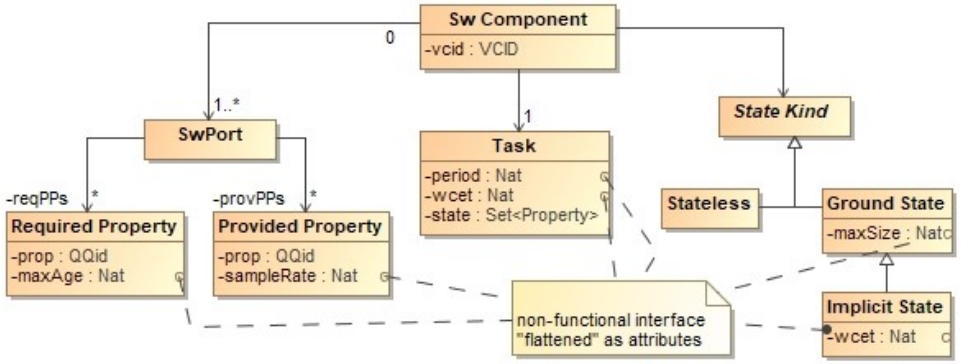

FIGURE 3.2: An overview of the component type model in UML. A versioned software component type has a unique versioned component identifier (VCID), a set of software ports, one task declaration and its state kind. An important part of the component type is its non-functional interface (e.g., the period of the task).

which is not part of the model fragment in this section, as it results from the runtime model. The component implementation itself is a blackbox to integrators and the runtime environment, but needs to be provided by the component developer before execution. The implementation is basically given by a function defining the translation of inputs to outputs, possibly including an internal state. For reconfiguration purposes, a minimal optional reconfiguration interface can be specified and implemented in case state transfer should be supported.

Component Description

Figure 3.2 shows an overview of the proposed structural meta-model of software components in UML notation. In essence, a software component has software ports with required and provided properties, one task, and potentially a state. For simplicity, the non-functional properties (*maxAge*, *sampleRate*, *period*, *wcet*, *maxSize*) are "flattened" into this model as attributes. A more mature model would provide more extensibility and reusability. For instance, in the NFP modeling framework of MARTE [OMG19], UML elements can be freely associated with non-functional property (NFP) instances. However, this freedom in turn would cause meta-modeling complexity in our formal approach, especially dealing with constraints (only allow certain NFPs for specific classes), while we currently need a focused, minimum viable non-functional model. The following definitions specify the structure of software components in Maude. We define versioned component type identifiers consisting of two strings – a component type identifier and a version identifier:

```
sort VCID .
op __ : String String -> VCID [ctor] .
```

The versioned component type is given as "*c v*" with a whitespace in-between, e.g. *ButtonController 1.0*. In more algebraic definitions the versioned component type identifier $C = (c, v)$ is denoted as c^v, e.g. $c^{1.0}$. We choose the version identifiers from $\mathbb{N} \times \mathbb{N}$ and denote them with a period in-between as "M.m", e.g. "1.0" for major version $M = 1$ and minor version $m = 0$. Throughout this specification, no semantics is employed over the versioning other than unambiguously identifying a specific version of a component type. Hence, both the component type identifier and the version identifier could also be arbitrary strings or use another versioning scheme, e.g. Semantic Versioning [Pre13], i.e., <major>.<minor>.<patch>. The <major>.<minor>

versioning scheme is used, because it is more compact and still illustrates the versioning idea. Future conceptual extensions could introduce semantics regarding behavioral compatibility based on versioned component types. We define interface compatibility based on data types and timing. Therefore, the Maude code in listing 3.1 specifies a minimal type system to declare data types. The primary focus of

```
sort Type .
ops bool uint32 uint64 : -> Type [ctor] .
op struct : NzNat -> Type [ctor] .
op size : Type -> NzNat .
```

LISTING 3.1: The data types used in this specification.

this minimal type system is to analyze non-functional runtime behavior, e.g. the duration of transmitting values over network. Additionally, functional compatibility is required when connecting ports, such as software ports. For each Type t, there exists a positive natural number $size(t)$, which specifies the bit size of values of this type in memory and on wire. We define the primitive types $\mathbb{T}_p = \{bool, uint32, uint64\}$. Additionally, structures $\mathbb{T}_s = t_1 \times \cdots \times t_i$ of types $t_1, ..., t_i \in \mathbb{T}_p, i \in \mathbb{N}$ can be defined, the sizes of which are the sum of their components' sizes, i.e., their Type is $struct(\sum_i size(t_i))$.

Listing 3.2 specifies qualified quoted identifiers, properties and software ports. A software port has a provided and a required interface. The required interface of the software port is a set of required (input) properties declared by type, name and the required $maxAge \in \mathbb{N}$, the maximum age (us) allowed until a fresh value is required. The provided interface of the software port is a set of provided (output) properties declared by their type, name and the provided $refreshmentPeriod \in \mathbb{N}$, the period (us) in which the property is updated. The *maxAge* and *refreshmentPeriod* belong to the non-functional interface of the component. The idea is, that all non-functional aspects required and provided by a component to do its work must be declared explicitly, so that the engineering system and the runtime environment can automatically solve or at least detect non-functional issues and help to reduce the complexity of composition, deployment and reconfiguration. The non-functional interface defined in this specification is minimal and primarily serves as a first break-through to show how non-functional component interfaces can be used to support the goal of seamless third-party integration in distributed real-time systems. To improve the readability and compactness of port definitions, we sometimes collapse provided and required interfaces to one set of properties. To indicate the direction of properties, required (input) properties $t\ pp \mapsto n \in reqIf$ are then written as ${}^{\triangleright}pp_t^n$ and provided (output) properties $t\ pp \mapsto n \in provIf$ are written as ${}^{\triangleleft}pp_t^n$. Additionally, we use units to make higher numbers more readable.

```
sorts QQid MQQid .
subsorts Qid MQQid < QQid .
op _._ : Qid QQid -> MQQid [ctor prec 1] . --- qualified quoted identifiers 'x.'y.'z
sort Property .
op __ : Type QQid -> Property [ctor] .
class SwPort | reqIf: Map{Property,Nat}, provIf: Map{Property,Nat} .
```

LISTING 3.2: Model of software ports: required Properties are mapped to a maxAge and provided Properties to a refreshmentPeriod.

A task declaration consists of a state declaration, the requested period of task invocations, and the WCET of the task in microseconds:

```
class Task | fields: Set{Property}, period: Nat, wcet: Nat .
```

LISTING 3.3: Specification of tasks in Maude.

The state declaration of a task is a set of state properties (also referred to as the internal state of the component). The task only declares internal state needed for the computation, which corresponds to internal variables of the program; these cannot be accessed from the outside of the component without low-level knowledge about the program's layout in memory. Thus, during reconfigurations the component has to help actively to transfer this internal state. However, there are different notions of 'state' in this model. Listing 3.4 shows the different state kinds, which also declare how state transfer is supported (if the component is stateful at all). A component can be stateless, i.e., it works without a state or does not need state transfer during reconfigurations to stay consistent. It can have a ground state, i.e., the component implements the ground state pattern[2]: The component automatically considers and updates a persisted state in each task invocation, so only this state needs to be transferred during reconfigurations. Finally, it can have implicit state kind, i.e., the component has an internal state as described above, which cannot be accessed directly, but the component implements a special dump/load API for creating/consuming a ground state on demand (called state dump). The component implementation (including dump/load) is treated later in this section. If the state kind is ground state, then it specifies the worst-case bit size of the ground state. This is needed to calculate the time needed for its transmission, for instance. If it is implicit state, then it specifies the worst-case bit size of the state dump. Additionally, the WCET needed for dumping or loading the internal state to or from a state dump must be declared for implicit-state components.

```
sort StateInfo .

op stateless : -> StateInfo [ctor] .
op groundState : Nat -> StateInfo [ctor] .         --- maxSize (bit)
op implicitState : Nat Nat -> StateInfo [ctor] .   --- maxSize (bit), wcet (us) for dump/load
```

LISTING 3.4: The state kinds of software components.

A (versioned) component type is given by the versioned component type's identifier, the communication interface as set of named software ports, one task declaration, and its state interface:

```
class SWC | meta: VCID, ports: Set{Object}, task: Object, kind: StateInfo .
```

LISTING 3.5: Definition of (versioned) software component types.

Each versioned component type identifier c^v is only used as *meta* information of one versioned component type. A software component provides one task, which processes data received via required properties of its ports (inputs) in a cyclic manner and sends calculated data via provided properties of its ports (outputs), possibly including a state. The information needed to compose and deploy software components are all provided in the component type description – no knowledge about its implementation is needed. This is accomplished by adding non-functional information to the component type description, which is currently: Bit sizes of properties and state, WCETs of tasks and the dump/load functions, the task's period and (required/provided) sample rates of properties. In this thesis, both *maxAge* and *refreshmentPeriod* are mostly equal to the period of the single task of a component. In [TSK18] we described how the two sample rate values can be used to produce or consume multiple values per cycle. However, if the period and sample rates were not equal, the definitions of the task implementation, especially of the value assignments, needed to be adapted accordingly to represent multiple values, which we

[2]The ground state pattern is described in detail by Kopetz [Kop11].

omitted in this specification for simplicity. Another reason for the distinction between sample rates and periods is the potential to extend this component model towards multiple tasks per component or compositional components running with different periods using different subsets of the component's properties.

Example 1. *Following Maude code defines a versioned component type for a button controller with no internal state, a period of 50 milliseconds, a WCET of ten milliseconds and ground state size of 100 bits:*

```
op btn-v1 : -> Object .
eq btn-v1 = < 'btn-v1:SWC |
  meta:  "Button Controller" "1.0",
  ports: < 'btnIn:SwPort | provIf: empty, reqIf: bool 'btnState |-> 50000 >,
         < 'btnPort:SwPort | provIf: bool 'risingFlank |-> 50000, reqIf: empty >,
  task:  < 'checkButton:Task | fields: empty, wcet: 10000 >,
  kind:  groundState(100) > .
```

In algebraic notation the ports would be denoted as follows:

$$btnIn = \{^{\triangleright} btnState^{50ms}_{bool}\}, \tag{3.1}$$

$$btnPort = \{^{\triangleleft} risingFlank^{50ms}_{bool}\}. \tag{3.2}$$

Component Implementation

To complete the definition of a software component, the implementation is needed. Again, it is important to note that the component implementation can be defined independently from other steps such as deployment and reconfiguration. It is of course also possible to define a versioned component type after its implementation (e.g. for re-using an already existing implementation), if it complies with the following implementation abstraction. The implementation is specified by functions, which consume and produce values of the data types specified beforehand. Thus we first define values and corresponding functions in listing 3.6:

```
sorts NtVal Val .
subsort NtVal < Val .
class SerializableObject | size : NzNat .

op vNull : Time -> Val [ctor] .
ops vBool val32 val64 : Nat Time -> Val [ctor] .
op vObj : Object Time -> NtVal [ctor] . --- SerializableObject

op type : Val -> Type .
op age : Val -> Time .
op vAging : Val Time -> Val .
```

LISTING 3.6: The definition of values along with corresponding functions.

Each value has a parameter of sort *Time*, which can be retrieved via the *age* function. The age is set to zero when the value is created (e.g. an output computed during the task execution) and is increased using *vAging* as time passes. The value's data parameter depends on the *type*. For primitive types $t \in \{bool, uint32, uint64\}$, the constructors *vBool*, *val32* and *val64* create corresponding values, where the data parameter is a natural number within the representable range:

- $vBool = \{false, true\} = \{\bot, \top\} = \{0_b, 1_b\}$
- $val32 = \{0_{u32}, \ldots, 2^{32}_{u32}\}$
- $val64 = \{0_{u64}, \ldots, 2^{64}_{u64}\}$

Subscripts are used to indicate bit lengths of numeric values, but they can be omitted for brevity. The standard operations ($+$, $-$, ...) for numeric values and logical operators ($\neg$, &, ...) are used as defined in C18 [ISO18]. The data parameter of

a structured type is given by objects of user-defined classes derived from the base class *SerializableObject*. The bit size of such a value is kept by its *size* field, and the exact type is $struct(size)$. The constructor function *vObj* constructs a value from such an object. The size of a value v can be retrieved with $size(type(v))$ in general. Finally, *vNull* is a typeless value, which can be used instead of proper values if no data is set (e.g., when an input is missing). No further operations are defined for *vNull*.

```
class ComputingContext | inputs: Map{Property,Val}, memory: Map{Property,Val},
  outputs: Map{Property,Val}, rte: Config .
--- computes outputs from inputs based on ComputingContext Objects
op doTask : VCID Object -> Object .
--- creates a StateDump object from the value assignment for the internal state
op doDump : VCID Qid Map{Property, Val} -> Object .
--- creates an updated value assignment for the state properties from a StateDump object
op doLoad : VCID Object -> Map{Property, Val} .
```

LISTING 3.7: Function prototypes for the component-specific implementation of its task as well as dump and load for implicit-state components.

Listing 3.7 shows the functions which define a component's implementation. The task implementation of a versioned component type c^v is given by an equation for the *doTask* function, the VCID parameter of which matches c^v. The function takes a *ComputingContext* object (an abstraction of *AppContainers*, see section 3.2), which holds value assignments for the input properties (*inputs*), state properties (*memory*) and ground state objects (inherited from *SerizalizableObject*) within the *rte* Configuration. Depending on the state kind, any of these fields may be empty. The equation provided by the component developer yields the modified *ComputingContext* as of the logic of the component. It should update the state properties and the ground state objects as well as the value assignment for the output properties (*outputs*). Similarly, the two functions *doDump* and *doLoad* provide a translation between the component's internal state and *StateDump* objects (inherited from *SerializableObject*). It is important that the function definitions comply with the state kind of the component. Stateless components are assumed not to depend on or provide a non-empty ground state or a non-empty internal state. Hence, dump and load need not be defined, as stateless components do not need to dump or load state during reconfigurations. For ground state components, the cycle's initial *memory* of the *ComputingContext* should be ignored, as it should either be loaded from the current ground state or not be used at all. The resulting internal state should also be ignored and persisted to a ground state object within *rte*. It is assumed that the ground state is always updated and used so that potential reconfigurations can handle the state transfer based on the ground state, only. Thus, dump and load are not needed during updates (however, these functions may be used to define the *doTask* equation(s), if needed. Only for components with internal states, the *doDump* and *doLoad* functions serve as reconfiguration interface and are not used for normal task implementation. The different handling of state transfer during reconfigurations are described in chapter 4.

Example 2. *Listing 3.8 defines the implementation for the ground-state component type* $ButtonController^{1.0}$ *from example 1.*

```
class BoolStateDump | val: Bool .
subclasses BoolStateDump < StateDump < SerializableObject .
eq doTask("Button Controller" "1.0", <C:ComputingContext|
    rte: < 'prev-in:BoolStateDump | val: B-PREV-IN >,
    inputs: bool 'btnIn.'btnState |-> CURR-IN >)
= <C:ComputingContext| rte: < 'prev-in:BoolStateDump | val: asBool(CURR-IN) >,
    outputs: bool 'btnPort.'risingFlank |-> CURR-IN and not vBool(B-PREV-IN) > .
```

LISTING 3.8: The task implementation of $ButtonController^{1.0}$ using a ground state.

Concluding, we specified the structure and implementation of software components. Software components, according to our model, process data in a deterministic manner, both functionally and non-functionally. Component types give a clear interface about expected and provided data, processing resources, and timing. The component implementation is only needed for execution simulation. Only the component type declaration (i.e., no component implementation) is needed for composing, deploying and reconfiguring software components at design-time as described in the following sections.

3.1.2 Distributed Embedded Applications

The functionality of a distributed control system is provided by a distributed embedded application. A distributed embedded application consists of component instances and connectors, which wire the components instances' ports to define communication relationships between them. Not all ports need to be connected – they can also be mapped to I/Os in a later step or be left unmapped. However, all required properties must be provided in the end. A distributed embedded application has a global period, which applies to all its components, i.e., to the period of tasks and to sample rates. The runtime behavior of a distributed embedded application is specified in later sections, but at this point it is worth noticing that all tasks are executed synchronously according to the logical execution time paradigm [KS12]: Input properties are first provided to all components, then their tasks are triggered and after the period is over, the output properties are extracted and further handled. Finally, the end-to-end reaction time of a distributed embedded application from inputs to outputs (including computation and communication) can be constrained. The fulfillment of this constraint can only be answered definitely after the design is complete, especially after we know which component runs on which node. The logical execution time paradigm helps to keep this and further non-functional analysis comprehensible (see section 3.3) as it provides temporal determinism while avoiding detailed scheduling configuration (e.g. priorities, time slices, ...). Thus, fixed logical execution time simplifies the composition of components by third-parties. Additionally, the synchronous cycle turnover is a global quiescent state of a distributed embedded application and enables the reconfiguration concept specified in chapter 4.

```
sort Matching .
op _-->_ : QQid QQid -> Matching [ctor] . --- [[swc.]port.]prop --> [[swc.]port.]prop
op _<--_ : QQid QQid -> Matching [ctor] .

class Connector |
  ports:    Tuple{QQid,QQid}, --- qualified SwPorts (SWC.port)
  matching: Set{Matching}.    --- Properties of the Ports (prop)
```

LISTING 3.9: Definition of property matchings and port connectors.

A component instance is given by an application-unique component instance identifier and the component's interface – a versioned component type (identifier). The versioned component type $ButtonController^{1.0}$ defined in example 1 is already a prototype for a component instance of this type. Thus a component is instantiated by creating a copy of the versioned component type and setting the object identifier according to the component instance identifier. Consequently, the task and the ports are copied, too (along with all required and provided properties). They are identified using qualified quoted identifiers (QQid) to avoid name clashes. Multiple instances of a component type can exist in a distributed embedded application. Listing 3.9 shows the definition of port connectors. A port connector (connector)

connects properties of the two software ports $c_i.p_n$ and $c_j.p_m$ given in its *ports* tuple. QQids are used, where p_n identifies a port of component instance c_i and p_m a port of component instance c_j. Additionally, the connector specifies a set of matchings $(c_i.p_n.pp_x, c_j.p_m.pp_y)$ between properties of port p_n and properties of p_m The meaning of a matching $(c_i.p_n.pp_x, c_j.p_m.pp_y)$ is that the value of a provided property $c_i.p_n.pp_x$ shall be extracted after each period, possibly transmitted to the location of c_j and finally injected to the required property $c_j.p_m.pp_y$, or the other way round, if pp_y is provided and pp_x is required (referred to as the direction of the matching or data flow). The matchings are used enable independent naming of properties in two connected components, so they don't need to use the same 'language', but can call the properties according to the vocabulary used in the corresponding domain or discipline. On both sides of a valid connector there must be exactly one matching for each required property with a provided property of the same type. Depending on the direction of a matching, we write a matching $(c_i.p_n.pp_x, c_j.p_m.pp_y)$ as

$$c_i.p_n.pp_x \rightarrow c_j.p_m.pp_y, \qquad \textit{if the data flow is from } pp_x \textit{ to } pp_y, \tag{3.3}$$

$$c_i.p_n.pp_x \leftarrow c_j.p_m.pp_y, \qquad \textit{else}. \tag{3.4}$$

A distributed embedded application (*eApp*) is given by a set *swcs* of component instances, a set *conns* of valid connectors within *swcs*, where each required property is matched at most once, the global cycle *period* in microseconds, and a constraint on the worst-case reaction time *wcrt* in microseconds as specified in listing 3.10:

```
class eApp | swcs: Set{Object}, conns: Set{Object}, period: Nat, wcrt: Nat .
```

LISTING 3.10: Definition of distributed embedded applications.

This way an *eApp* puts together all the application-layer pieces of a distributed control system. It has a set of component instances, where each instance has one task, the *period* of which must be set to the global period of the *eApp*. Obviously, the tasks' WCET *wcets* must be smaller than the *period* of the *eApp*. The component instances' ports are wired with connectors, so that inter-component communication is already defined on a logical level. The *wrct* parameter of an *eApp* constrains the worst-case reaction time of the final distributed control system from sensors to actuators. While further schedulability analysis depends on the deployment, we can already make a first feasibility analysis of the worst-case reaction time: Due to the logical execution time paradigm, each processing of an input property by a task will take exactly the time of one period. The worst-case communication time of a connector between co-located component instances is zero (due to the zero execution time assumption of the cycle turnover). Hence, a lower bound of the worst-case reaction time can be calculated from the number of tasks involved in the dataflow from sensors to actuators. We have to find the longest path from an input property to an output property in the dataflow graph (the construction of which is described in section 3.3); if the path passes n tasks, then the minimum possible worst-case reaction time of the *eApp* is $(2 + n) \cdot period$ (plus two periods because of the sampled inputs and outputs), and we can check if it is smaller than *wcrt*. After the deployment, we can take into account the real communication and sampling times.

Example 3. *The software part of the onBtnSwitch example is a distributed embedded application as defined in listing 3.11.*

```
op led-v1 : -> Object .
eq led-v1 = < 'led-v1:SWC | meta : "LED Controller" "1.0",
  ports:  < 'ledPort:SwPort | provIf: empty, reqIf: bool 'switchCmd |-> 150000>,
          < 'ledOut:SwPort | provIf: bool 'ledState |-> 50000, reqIf: empty >,
  task: < 'setLed:Task | fields: empty, wcet: 10000 >,
  kind: groundState(100) > .
op onBtnSwitchEApp : -> Object .
eq onBtnSwitchEApp = < 'onBtnSwitchEApp:eApp |
  swcs : btn-v1, led-v1,
  conns: < 'btnLedCon:Connector |
      ports : ('btn-v1.'btnPort, 'led-v1.'ledPort),
      matching : 'risingFlank --> 'switchCmd >,
  period: 50000, wcrt: 250000 > .
```

LISTING 3.11: An *eApp* for the *onBtnSwitch* example system including the definition of the *LED Controller*$^{1.0}$. The *Button Controller*$^{1.0}$ from example 1 ist used via the *btn-v1* operation. A connector connects two ports of the two components, so the *risingFlank* output of *btn-v1* is used as input for *led-v1*. The global period is 50 ms and a worst-case reaction time requirement of 250 ms is specified, which is satisfiable, because two tasks are involved in the dataflow: $(2{\cdot}2){\cdot}50\,\text{ms} < 250\,\text{ms}$.

These definitions make up a minimal viable model of distributed embedded applications for the purpose of this thesis: We study simplified and evolvable third-party integration of distributed embedded applications towards the extreme of continuous updates during full operation despite end-to-end real-time requirements. Obviously, it is not the most comprehensive application model and has potential for various future extensions. For example, components are not defined compositionally, but they are composed in a flat manner. Additionally, each component only has one task. The ports only support properties, but no operations or events. Both task execution and inter-component communication are aligned in an isochronous global cycle. And finally, the non-functional specification is fairly minimal, e.g. we specifiy only a global worst-case reaction time constraint instead of more fine grained bounds. However, the model is still comprehensive, as we have to take these distributed embedded applications all the way to the execution environment in subsequent model fragments to finally analyze the non-functional behavior and to describe and perform even compatibility-breaking reconfigurations without downtime (or at least with deterministic temporary quality degradation). Therefore, the syntactic and semantic simplicity of distributed embedded applications is a feature, as it allows us to model the complete chain.

3.1.3 System Models and Topologies

This section is on the hardware part of distributed control systems and on how the involved MCUs are connected with each other by means of communication media and to equipment by means of I/O ports. Thus we first introduce a model of MCUs, the MCU description[3], which is used to model an MCU's hardware components. Then network topologies of such MCUs can be described, which make up a distributed system to which distributed embedded applications can be deployed.

An MCU desciption is given by its unique identifier and the hardware components installed on the board (see listing 3.12 and figure 3.3). The hardware components are either network ports (currently only ethernet ports) or I/O ports (currently only general-purpose input/output ports (GPIOs)). For network ports a natural

[3] "MCU description" is our original wording from the master's thesis [Tel15], which is more self-explaining, as it describes an MCU. In intermediary work we referred to this model element as "board" [TSK18].

```
class IoPort | swPort: Object . --- the I/O interface as SwPort
class NetworkPort | bps: Nat .
classes HwComponent EthPort GPIO .
subclasses GPIO < IoPort < HwComponent .
subclasses EthPort < NetworkPort < HwComponent .
class McuDescription | hwComponents: Set{Object} .
```

LISTING 3.12: MCUs defined by the set of installed hardware components.

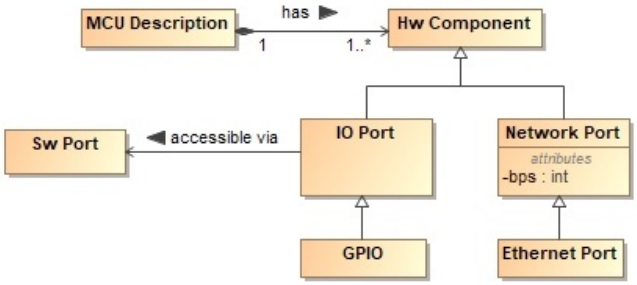

FIGURE 3.3: The MCU description model in UML. An MCU consists of two types of hardware components: I/O ports for accessing equipment and network ports for communication with other MCUs.

number *bps* must specify the bits per second of guaranteed bidirectional throughput the network port is capable of. For I/O ports a software port must be specified, which defines the API through which software components can access the equipment by means of the execution environment. In an engineering tool, this API would not be freely specified, but it would be selected and configured according to a predefined set of I/O interfaces associated with corresponding hardware component types on an MCU. Due to the limited scope of this specification, we keep the I/O model simple and only define GPIO inputs and outputs, which are accessible via one (required or provided) boolean property for digital I/Os or one (required or provided) integer property for numeric-valued I/Os, as shown in the following example. It is up to future work to define more complete sets of I/O interfaces and to enable these interfaces in a generic way as we did for GPIOs[4].

Example 4. *The MCU IOT2040 can be modelled as as shown in listing 3.13.*

```
op IOT2040 : -> Object .
eq IOT2040 = < 'IOT2040:McuDescription | hwComponents:
  < 'gpio13:GPIO | swPort: <'usrLed:SwPort| provIf: bool 'ledOut |-> 10000, reqIf: empty> >,
  < 'gpio63:GPIO | swPort: <'usrBtn:SwPort| provIf: empty, reqIf: bool 'btnIn |-> 10000> >,
  < 'eth0:EthPort | bps: 10000000 >, --- 10 Mbit/s
  < 'eth1:EthPort | bps: 10000000 > > .
```

LISTING 3.13: The SIMATIC IOT2040 modeled with its two ethernet ports, the user button and the user led.

In more compact algebraic notation the I/O interfaces are:

$$gpio13 \mapsto {}^{\triangleleft}ledOut^{10ms}_{bool} \qquad \textit{and} \tag{3.5}$$

$$gpio63 \mapsto {}^{\triangleright}btnIn^{10ms}_{bool}. \tag{3.6}$$

The LED output in example 4 is accessible as a provided property with the QQid *gpio13.usrLed.ledOut* and the button input as a required property *gpio63.usrBtn.btnIn* (however, as I/O ports only have one property in this thesis, we sometimes omit the port or the property). The runtime handles these interfaces as delegation ports,

[4] AUTOSAR is a good example for how complex this problem actually is. For example, consider the specifications for I/O Hardware Abstraction [AUT19b] and ECU Configuration [AUT19a].

i.e., the values are forwarded between the I/O ports and possibly mapped ports of locally installed software components. This is described more detailed in the subsequent section on deployments. Now that we have MCU descriptions, we can use them to define nodes and the network topology between them. This is effectively a bidirectional graph of nodes, where edges are given by network connections. Distributed embedded applications are deployed to such topologies with minimal additional effort and with no manual system configuration required after design-time.

```
class NetIf | port: Qid, ipAddr: String, macAddr: String .
class Node | mcu: Object, netIfs: Set{Object} .
```

LISTING 3.14: Definition of nodes and network interfaces.

Listing 3.14 defines nodes and network interfaces. A network interface is described by a hardware component identifier (of an ethernet port of the corresponding MCU), a system-unique IP address, and a unique MAC address. IP addresses and MAC addresses are denoted in their usual octet-notation (e.g. "192.168.200.1" and "de-ad-be-ef-fe-ed"). This minimal platform-specific information is added at this point to keep all required information together in one pragmatic model of the nodes and the topology. While it is not important during modeling, it is essential for configuration of the nodes, later-on. To add support for different communication media and ports (CAN, Wi-Fi, ...), specific description types must be added in future as well as additional types of network connections. Network interfaces are used in the definition of nodes, as also shown in listing 3.14. A node description is given by the node identifier, the MCU description *mcu*, and a set *netIf* of network interface descriptions for the network ports of *mcu*. The nodes are meant to be instances of MCUs, but it is more compact to put the MCU description into the node description and prune the MCU description to the relevant hardware components. For instance, if we don't use all CPU pins on a specific node, we only include the used GPIOs in the corresponding node description. Thus we may use two different MCU descriptions for two nodes of the same MCU type, which has the benefit that we can reduce the size of the resulting runtime image. There must be at most one network interface description for each network port of the MCU.

```
class NetConn |
  netIfs: Set{QQid},  --- two NetIfs (node1.if, node2.if)
  bps: Nat.           --- bidirectional throughput
```

LISTING 3.15: Definition of network connections.

A network connection between two network interfaces of different nodes can be specified as shown in listing 3.15. A network connection is described by a system-unique network connection identifier, the qualified identifiers of the two conneted network interfaces, and the bidirectionally guaranteed throughput *bps*. Such a network connection description specifies that there exists a direct network connection between the Ethernet ports associated with the network interfaces (via their *port* field). Note: To keep the model comprehensive we don't include special model elements for network-only nodes (switches or routers) in between. However, we will have to configure both ARP and IP statically to avoid discovery protocols. The easiest solution is to model them using node descriptions without mapping software components to them. In this way we have the information for configuring routes and firewalls, but we need special treatment for the network behavior, as usually each hop costs one period transmission time due to the logical execution time model. Hence, for configuration purposes, we need the topology as defined below including network-only nodes, and for all other purposes (such as simulating or reconfiguring a deployed distributed embedded application) we use a "flattened"

view of the topology, which does not include network-only nodes and replaces indirect network connetions $node_1 \leftrightarrow switch_1 \leftrightarrow node_2$ with direct network connections $node_1 \leftrightarrow node_2$. For network configuration and feasibility analysis, the *bps* parameter of network connections is used, which specifies a lower bound of the bidirectional throughput. It must consider the throughput of the two Ethernet ports, but also reflect abstractions such as flattening the network topology as described beforehand. In the scope of this thesis we do not handle network errors, but consider the throughput as-is and instantly issue errors if network messages are missing. We strongly suggest that either the platform or software components should be extended to detect and compensate transient network faults.

```
class Topology |
  nodes: Set{Object}, --- Nodes
  conns: Set{Object}. --- NetConns
```

LISTING 3.16: Specification of network topology descriptions.

Finally, we put together the pieces by specifying network topologies (see listing 3.16). A network topology is described by a set *nodes* of node descriptions (*Node* objects) and a set *conns* of network connections between the nodes (*NetConn* objects). A network topology description is a bidirectional graph between nodes, which is induced by network connections. We have already elaborated on network-only nodes beforehand. At this point it is worth mentioning that the network topology includes the engineering system, for instance, because software components need to be downloaded during reconfigurations as described in chapter 4. The following example for the *onBtnSwitch* system includes such an engineering system and network connections for administration communication.

Example 5. *Listing 3.17 defines a (flattened) network topology description for onBtnSwitch consisting of two IOT2040 MCUs and one engineering system connected via Ethernet (see figure 3.4). Each of the three nodes node1, node2 and the engineering system ES is connected to each other node. The two 'real' nodes node1 and node2 are connected via their eth1 interfaces, which is the only connection of the plant network. They are connected to the engineering system via their eth0 interfaces, which belong to the administration network. This topology is used throughout the rest of this thesis if not stated otherwise.*

```
op twoIOT2000s : -> Object .
eq twoIOT2000s = < 'twoIOT2000s:Topology |
  nodes :
    < 'node1:Node | mcu: IOT2040, netIfs:
      < 'eth0:NetIf | port: 'eth0, ipAddr: "192.168.200.1", macAddr: "af-fe-af-fe-af-fe" >,
      < 'eth1:NetIf | port: 'eth1, ipAddr: "192.168.1.1", macAddr: "01-23-45-67-89-ab" > >,
    < 'node2:Node | mcu: IOT2040, netIfs:
      < 'eth0:NetIf | port: 'eth0, ipAddr: "192.168.200.2", macAddr: "de-ad-be-ef-fe-ed" >,
      < 'eth1:NetIf | port: 'eth1, ipAddr: "192.168.2.1", macAddr: "aa-bb-cc-dd-ee-ff" > >,
    < 'ES:Node | mcu: ZBook, netIfs:
      < 'eth0:NetIf | port: 'eth0, ipAddr: "192.168.200.100", macAddr: "fe-dc-ba-98-76-54" > >,
  conns :
    < 'con1 : NetConn | netIfs: 'node1.'eth1, 'node2.'eth1, bps: 1000000 >
    < 'admConn1:NetConn | netIfs: 'node1.'eth0, 'ES.'eth0, bps: 1000000 >
    < 'admConn2:NetConn | netIfs: 'node2.'eth0, 'ES.'eth0, bps: 1000000 > > .
```

LISTING 3.17: A network topology consisting of two IOT2040-based nodes and an engineering system. Each network connection has 1 Mbit/s set as bidirectionally guaranteed throughput

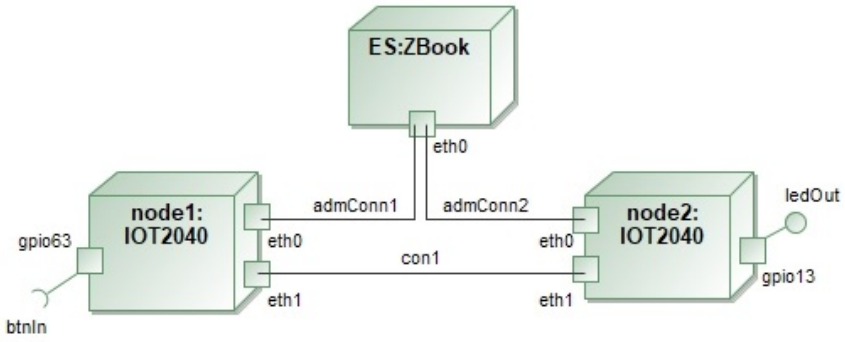

FIGURE 3.4: An UML diagram of the network topology *twoIOT2000s*.

3.1.4 Deployment Descriptions

After modeling a distributed embedded application and a network topology, a mapping between these two independent models can be defined – the deployment desciption. We have to distinguish between the deployment description as such and the process of deploying the distributed embedded application to the distributed system. The deployment as procedure is a reconfiguration and not covered in this section. This section describes the deployment description as such. It maps software to hardware and thus defines the desired system configuration (which can be obtained by reconfiguring accordingly, depending on the initial system configuration).

```
class Mesg | size: Nat .
class LocalMesg | size: Nat, node : Qid .
class NetMesg | size: Nat, sender: Qid, receiver: Qid, path: List{Qid}.
subclasses LocalMesg NetMesg < Mesg .
class ComSpec | msgs: Set{Object} . --- Mesg objects
```

LISTING 3.18: A communication specification defines the set of messages in the distributed system. There are node-local messages and 'real' network messages.

Listing 3.18 shows the Maude definition of communication specifications. A communication specification defines a set of messages for a network topology. It is defined during the deployment step to describe the physical communication needs in the network topology, complementing the logical communication needs specified by the connectors of the *eApp*. An abstract message (*Mesg*) only declares its *size*, i.e., the maximum number of bits to transport during a period. There are local messages (*LocalMesg*), which additionally have a *node* parameter. Local messages are used for inter-component communication of co-located components (which run on the same node). Additionally, there are network messages (*NetMesg*), which declare the *sender* and *receiver* nodes and a *path* within the network topology from the *sender* to the *receiver*. The *path* is given by an ordered list of network connection identifiers, where consecutive network connections start/end at network interfaces of the same node. Network messages are transmitted periodically, but only take one hop per period to simplify the schedulability analysis as descibed later in this section and in section 3.3.

```
class Deployment |  eApp: Qid, topo: Qid, com: Qid,
                    swcMap : Map{Qid,Qid}, comMap: Map{PropConn,Qid}, ioMap: Map{QQid,Qid}.
```

LISTING 3.19: Deployment descriptions consist of a component mapping, a communication mapping and an I/O mapping for running a specific distributed embedded application on a specific network topology.

A deployment description (see listing 3.19) for a specific distributed embedded application *eApp* and network topology *topo* consists of a SWC mapping *swcMap*, a communication specification *com*, a communication mapping *comMap*, and an I/O mapping *ioMap*. In essence, the deployment description maps three kinds of software-related elements (from a distributed embedded application) to corresponding elements related to hardware and network (from a network topology description): The SWC mapping assigns a node in *topo* to each component in *eApp*. The communication mapping assigns a message from *com* to each matching, so that each matching $c_i.p_n.pp_x \rightarrow c_j.p_m.pp_y$ of the connectors in the *eApp* is assigned a message the *path* of which starts at c_i and ends at c_j as of the SWC mapping. The injective I/O mapping (i.e., I/Os are accessed exclusively) assigns a local I/O port *node.io* to each unmapped software port $c.p$ of the *eApp* (the I/O port *io* is local if c is deployed to that *node*). The three mappings have following meaning and implications:

The SWC mapping assigns a node to each software component of a distributed embedded application. The software components shall be installed to and executed on the assigned node. It will consume resoure shares from that node (especially CPU, Memory, Network) in competition with other co-located software components. Thus we have to make sure, that each node has enough computing resources to host the assigned software components.

The communication mapping assigns (network) messages to each matching of each connector of a distributed embedded application. While a connector is a logical communication channel which declares a data flow between provided and required properties, (network) messages specifiy the real communication channel as path in the network topology through which the properties are transmitted from sender to receiver, including possible intermediary hops. Multiple matchings can be assigned to the same network message, as long as they need to be transported across the same path and as long the message's bit size is sufficient. However, network messages are unidirectional, thus two network messages are needed to cover bi-directional connectors. To decrease the reconfiguration effort, only matchings of the same connector are assigned the same network message. Effectively, provided properties are extracted after each cycle, transmitted to the receiving software component and injected at deterministic instants (as described in the runtime model).

The I/O mapping assigns I/O ports to unmapped software ports of a distributed embedded application. Each I/O port of a node can only be assigned to one port of the software components it hosts according to the SWC mapping. The I/O port's software port (its API) and the assigned software component's port must be identical (apart from renaming). As we currently only support GPIOs, the software component's port must have exactly one required (for inputs) or provided (for outputs) property of the same type as declared in the I/O port. Required properties will be taken from the driver (ensuring the declared maximum age) and provided to the software component before each cycle. Provided properties are extracted from the software component after each cycle and forwarded to the driver.

Example 6. *Listing 3.20 defines a communication specification and a deployment description for the onBtnSwitch example system (see figure 3.5), i.e., a deployment of the distributed embedded application from example 3 and the network topology from example 5. The Button Controller btn-v1 is deployed to node1 and the LED Controller led-v1 is deployed to node2. The unmapped ports btn-v1.btnIn and led-v1.ledOut are mapped to I/O ports of the*

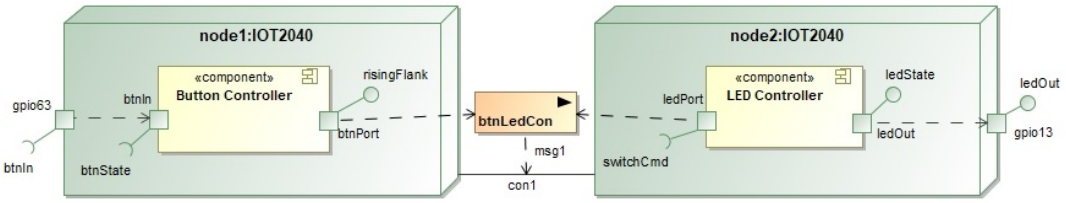

FIGURE 3.5: An UML diagram of the deployment *onBtnSwDep*.

corresponding nodes. The matching between the risingFlank output and the switchCmd input is mapped to the network message msg1, the path of which goes from node1 to node2 (via the network connection con1). For each network connection an additional network message for platform-level communication in both directions is specified, the quota of which depends on the parameter MsgAdmin.

```
op comSpec : -> Object .
eq comSpec = < 'comSpec:ComSpec | msgs :
  < 'msg1:NetMesg | size: 1, sender: 'node1, receiver: 'node2, path: 'con1 >,
  --- admin communication towards the nodes
  < 'es2node1:NetMesg | size: MsgAdmin, sender: 'ES, receiver: 'node1, path: 'admConn1 >,
  < 'es2node2:NetMesg | size: MsgAdmin, sender: 'ES, receiver: 'node2, path: 'admConn2 >,
  --- admin feedback towards ES
  < 'node1-2es:NetMesg | size: MsgAdmin, sender: 'node1, receiver: 'ES, path: 'admConn1 >,
  < 'node2-2es:NetMesg | size: MsgAdmin, sender: 'node2, receiver: 'ES, path: 'admConn2 >,
  --- reconf agent messages P2P
  < 'node1-2node2:NetMesg | size: MsgAdmin, sender: 'node1, receiver: 'node2, path: 'con1 >,
  < 'node2-2node1:NetMesg | size: MsgAdmin, sender: 'node2, receiver: 'node1, path: 'con1 >
> .
op onBtnSwDep : -> Object .
eq onBtnSwDep = < 'onBtnSwDep:Deployment |
  eApp: 'onBtnSwitchEApp, topo: 'twoIOT2000s, com: 'comSpec,
  swcMap: 'btn-v1 |-> 'node1, 'led-v1 |-> 'node2,
  comMap: 'btn-v1.'btnPort.'risingFlank --> 'led-v1.'ledPort.'switchCmd |-> 'msg1,
  ioMap: 'btn-v1.'btnIn  |-> 'gpio63, 'led-v1.'ledOut |-> 'gpio13 > .
```

LISTING 3.20: A communication specification and deployment description for the *onBtnSwitch* example system.

As a minor difference to the engineering methodology published in [TSK18], the communication specification is no longer a part of the system description, but instead defined during deployment. The main reason for this change is that the set of messages needed depends on the needs of the specific deployment and thus must be defined during the design of the deployment. Additionally, the communication specification is software-defined and subject to dynamic reconfigurations without modifying the network topology as such. Finally, we suggest that – in a future extension of the approach – it should be possible to generate a valid deployment automatically (including the set of messages). Another aspect of the communication specification is its level of abstraction. We chose a high level of abstraction to make this model extensible by future refinements. In the definition and the explanation above we only stated that the messages are sent and received at deterministic instants, but did not further elaborate on priorities and timeslices. We will do this in the following. Again, the primary aim of this thesis is to keep third-party integration as simple as possible, while fulfilling hard end-to-end real-time requirements. Thus, the most simplistic approach was chosen, which perfectly fits to the globally synchronous fixed logical execution time: Each network connection ("hop") of a message exactly costs one cycle, as the message is released just before a new cycle, transmitted during the cycle and further processed after the cycle is completed. Thus, the effective communication time is deterministic without assigning time-slices: It

is equal to the path length times the period. We can therefore transmit network packets at arbitrary times during a cycle, which can be achieved with "any" protocol and without further protocol-specific configuration. In the real-time container architecture described in section 5, this is achieved via traffic shaping, i.e., applying bandwidth control to network messages. Consequently, we can use standard udp/ip communication to implement this model for the scope of this thesis (instead of dealing with industrial, real-time-capable busses and protocols). On the other hand, this approach also introduces huge communication times, depending on the period and the deployment. In future approaches, additional, protocol- and medium-specific pieces of information could be added to the communication specification to achieve shorter communication times. These approaches should also consider how we can relax the task scheduling model so that messages known to arrive earlier can be processed earlier, while still keeping the deployment procedure as simple as possible.

```
ops AdminTime BgTime Ovhd MsgAdmin : -> Nat [ctor] .
```

LISTING 3.21: Platform-specific system parameters. These need to be specified/configured to check the feasibility of a deployment.

Finally, the feasibility of a deployment description depends on following additional, platform-specific system configuration parameters declared in listing 3.21:

- *AdminTime* ($WCET_{adm}$), the configured CPU quota for the administration phase in microseconds per period (also written as $WCET_{adm}$, because the quota should be as least as big as the WCET of the administration phase),
- *BgTime* (Q_{bg}), the execution time assigned to background workers in microseconds per period,
- *Ovhd* (O_{msg}), the protocol overhead in bits per message,
- *MsgAdmin* (M_{adm}), the configured network quota for platform-level communication in bits per period per connection.

These parameters specify details of the runtime, including platform-specific aspects, which must be provided partly to configure the runtime and partly for feasibility analysis. Thus, we declare and briefly explain these parameters here, though they are tethered to the runtime model, the reconfiguration model, and the platform-specific real-time container architecture as specified in the subsequent sections and chapters. The administration phase is the time between two cycles, in which the platform extracts outputs from the components, injects inputs and triggers the next cycle. Conventionally, this cycle turnover is performed under ZET assumption. In our approach, additional reconfiguration operations are performed and we must ensure that there is enough time for them, before starting reconfigurations. Therefore, $WCET_{adm}$ specifies the amount of time reserved for the cycle turnover. Additionally, Q_{bg} is reserved for time-consuming reconfiguration operations in background (e.g. starting a new component). Only the rest of the period ($period - WCET_{adm} - Q_{bg}$) can be assigned to components' tasks. For the communication, there are two additional parameters. The protocol overhead O_{msg} is an upper bound of the per-message overhead (e.g. caused by headers). Throughout this thesis each property is sent via a dedicated message, so the message overhead is only caused once per period. However, if a message is used for n properties, then an overhead of $n \times O_{msg}$ is caused. This must be considered in the feasibility calculation (see section 3.3). Finally, M_{adm} is the amount of network bandwidth to reserve for platform-level communication

(including reconfiguration-related communication such as downloads). This bandwidth is reserved bidirectionally on each network connection. When not stated otherwise, the system configuration parameters are:

$$WCET_{adm} = 10\,\text{ms}, \tag{3.7}$$
$$Q_{bg} = 5\,\text{ms}, \tag{3.8}$$
$$O_{msg} = 432\,\text{bit} \quad (\text{UDP } 8\,\text{B},\ \text{IPv4 } 20\,\text{B},\ \text{MAC } 18\,\text{B},\ \text{Eth } 8\,\text{B}), \tag{3.9}$$
$$M_{adm} = 5000\,\text{bit} \quad (\text{PTP} \approx 1000,\ \text{coordination} \approx 2000). \tag{3.10}$$

3.2 Runtime Model

The runtime model is basically an executable refinement of the design-time model. Figure 3.6 shows an overiew of the runtime model (the colors indicate how it corresponds to the design-time model). It consists of a green layer for application execution within containers, a grey system layer for scheduling, I/Os and networking and a blue layer in-between for the system execution management by an agent. Each software component is executed within an application container on the corresponding node. The additional background container is used for the reconfiguration purposes described in chapter 4. The agent is installed on each node and mediates between the application and system layer. This ensures that the distributed embedded application operates as modeled in the deployment description and that it is reconfigurable. The system layer provides execution and communication means which are shared by applications. To make sure that these are available to the demanded extent we must (besides only accepting feasible deployments) enforce both the functional and the non-functional interface of each software component: Enforcing the resource limits of each app ensures that the remaining share is available to the others. For instance, network connections are used for inter-component communication as well as inter-agent communication and must be bandwidth controlled along the route of a message. The rest of this section specifies the three layers.

3.2.1 Application Containers

Application containers are the sphere of influence for software components: They isolate a component so that it cannot communicate with the outside of the container in ways not actively enabled by the agent. Of course communication with other components and equipment is needed and can be achieved with shared memory, network communication, I/O interfaces and other inter-process communication means. In this model, communication can only be achived implicitly by reading and writing properties, which have been mapped to messages or I/O ports. The platform-specific model has to define how this is implemented for the desired technologies (e.g. UDP communication). Additionally, the isolation has to ensure the non-functional interface, especially w.r.t. CPU time, network bandwidth and memory. Following architectural properties of the application container specification are important not only to ensure the functional and non-functional consistency of a deployment, but also to enable the reconfiguration concept (cf. [TK19a]):

- Sender and receiver don't know each other: The communication relationships defined in the deployment description are not created by the components themselves.
- No hidden communication channels: Communication attempts beyond the defined mappings do not reach the outside/inside.

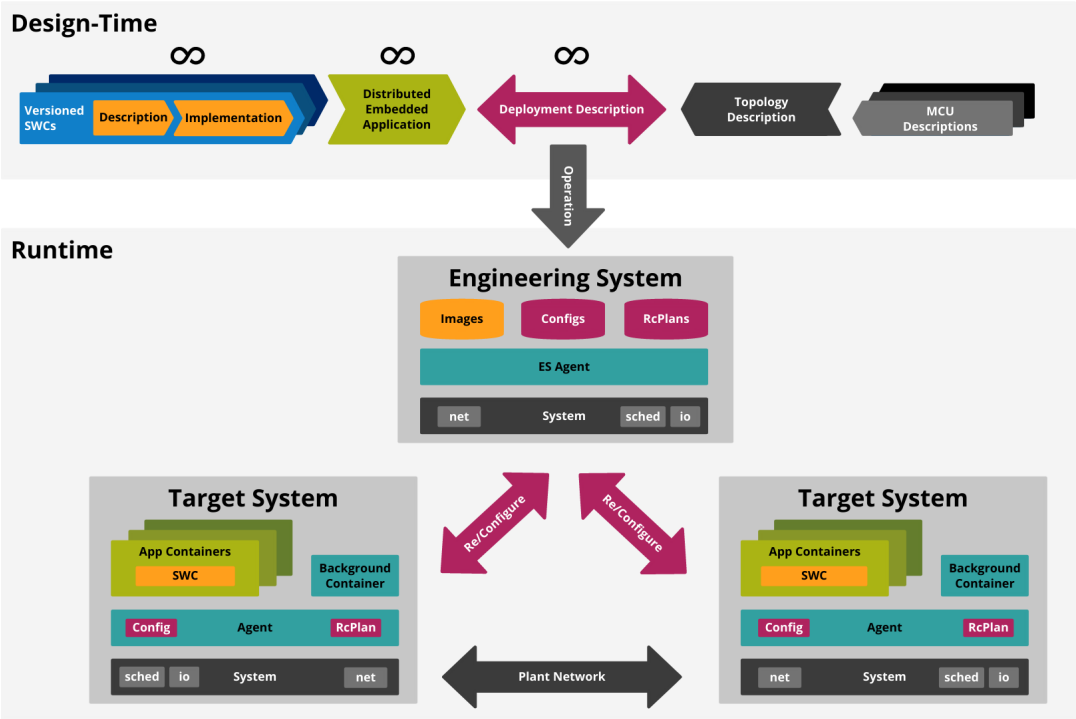

FIGURE 3.6: A schematic architectural overview of the runtime model. It consists of an application layer for component execution within dedicated containers, a system layer for scheduling, I/Os and networking and a management layer in-between for running the system accordingly on each node by means of an agent and accompanying containerized background workers. The design-time model is brought to operation statically via the agent configuration and/or dynamically via reconfiguration plans (reconfiguration is treated in chapter 4).

- Resource control: Shared resources such as network bandwidth and CPU quota are provided and limited as modeled, so deadlines are kept.
- Fixed logical execution time: A task can calculate its outputs from the inputs at any time until the deadline – inputs and outputs are decoupled from outside during a cycle.

Listing 3.22 shows the essential definitions of application containers (without reconfiguration aspects). A container has following states (*CState*): When newly created but not started or initialized it is *new*. After starting and initializing the container it is in *sleep* state. It is important that in sleep state the container is not only running but also the component inside is initialized and ready to perform a cycle. After triggering, the container is in *wait* state until it is assigned to a CPU and thus going to *run* state. It is possible to preempt containers during cycles, but the effects of context switches (e.g. time to restore, cache misses, etc.) are beyond the scope of this thesis. Thus, the proposed model assumes these effects are reflected by the WCETs of the tasks. As soon as the task completes or the container's quota expires, the container goes to *over* state and then back to *sleep* state when it has been removed from the CPU. Finally, after stopping the container, it goes to *stop* and can then be removed from the system.

Besides the state, a container has a *clock* field which measures the runtime received in the current cycle, a *wcet* field for the assigned CPU quota per cycle and a *counter* field which is incremented after each completed cycle. Additionally, a container has a *job* field, which is needed for reconfiguration purposes and described

later. Finally, a container has an *rte* field to hold all objects and messages needed by the task or job for its execution. For instance, a software component can store state information at this location. However, for hosting software components the *AppContainer* subclass is used, which has additional fields and a specialized behavior for this purpose. The *meta* field stores the versioned component type identifier of the component. *AppContainers* inherit three fields from *ComputingContext*, which map properties to values and are used to store the current valuations of input properties (*inputs*), of output properties (*outputs*), and of internal state properties (*memory*). The input and output properties are qualified by the port name. Finally, an *AppContainer* has a communication mapping field *comM*, which mapps qualified input and output properties to a message or I/O identifier (in the following, only message mappings are described, but the same applies to I/O mapped properties).

The behavior of (application) containers is defined by rules, which process specific messages, and by the *mte* and *timeEffect* equations, which modify the temporal behavior according to the system tick rule. An *AppContainer* c in state *sleep* updates the valuation of an input property $p.pp$ mapped to message m to the new value v after receiving the message $reveive(c, p.pp, m, v)$ (the same applies to I/O mapped properties). After receiving an $activate(c)$ message the container c changes the state from *sleep* to *wait* and resets its *clock*. Should the *clock* become equal to the *wcet* before the task or job running in the container is completed, the container runs out of CPU quota and changes to the *over* state to release the CPU. An *mte* equation ensures that time can only progress until the quota of a running container expires setting $mte = wcet - clock$. Otherwise, a completed task or job releases the CPU voluntarily by setting the container to *over* as in the [task] rule. This rule also modifies the output properties or the internal state of the container as of the implementation of the component; it is given by an equation for the *doTask* operation, which matches the versioned component type identifier *meta* (see section 3.1.1). A container in *over* state immediately transitions to *sleep*, increments the *counter* and sends a $done(c)$ message. This is enforced by another *mte* equation, which forbids progression of time if a container is in *over* state setting $mte = 0$. An output property $p.pp$ is extracted from a container c by sending a $getOutputCmd(c, p.pp, m)$ command message, which is responded by a c, m, v message containing the current valuation v of the requested output property, if it is mapped to message m in *comM*. Finally, *timeEffect* equations are defined so progression of time leads to aging of property valuations and if a container is in *run* state (i.e., it is assigned to a CPU) then its *clock* is updated, too.

```
ops new sleep wait run over stop : -> CState [ctor].
class Container | state:CState, clock:Time, wcet:Time, counter:Nat, job:Object, rte:Config.
class AppContainer | meta:VCid, comM:Map{QQid,Qid}.
subclasses AppContainer < Container ComputingContext.

rl[receive] receive(C,P,M,V) <C:AppContainer| sleep, comM[P]=M, inputs[TY P]=V'>
  => <C:AppContainer| inputs[TY P]=V)>.
rl[activate] activate(C) <C:Container| sleep, clock:T> => <C:Container| wait, clock:0>.
crl[quota] <C:Container| run, wcet:T, clock:T> => <C:Container| over> if not completed(<C|>).
rl[done]: <C:Container|over> => <C:Container| sleep, counter++ > done(C).
rl[send]: getOutputCmd(C,P,M) <C:AppContainer| sleep, comM[P]=M, outputs[TY P]=V >
  => <C:AppContainer|> send(C,M,V).
rl[task]: < SCRM:AppContainer | run, meta:NAME VERSION, job:NOP, wcet:US, clock:US >
  => doTask(NAME VERSION, < SCRM:AppContainer | over >) .

eq mte(<C:Container| run, job:_, clock:T, wcet:T'>) = T' - T.
eq mte(<C:Container| over>) = 0.
eq timeEffect(<C:Container| run, job:_, clock:T>, T') =  <C:Container| clock:T+T'>.
eq timeEffect(<C:AppContainer| inputs:I, outputs:O, memory:M>, T')
  = Container.timeEffect(<C| inputs:aged(I,T'), outputs:aged(O,T'), memory:aged(M,T')>, T').
```

LISTING 3.22: The specification of application containers in Maude (without reconfiguration).

Example 7. *Listing 3.23 shows an example of such an AppContainer for the Button Controller in the onBtnSwitch system. The state is set to new, the wcet is set to* 10 ms*, the counter and clock are at zero (so, this container has not performed a cycle, yet), and no job or rte objects are set. The three properties are all set to the default boolean valuation, which is false and has an initial age of* 0 us*. After being set to run state the [task] rule applies the doTask equation asociated with this container (through its meta information) when the clock reaches the value of wcet (i.e., after running for wcet microseconds). The equation (defined in listing 3.8) sets the risingFlank output to true iff. the current button input is set and has not been set in the previous cycle. The rest of the container execution is handled from outside by actively extracting outputs, injecting inputs and triggering the task at specific instants.*

```
eq btn-v1 = <"btn-v1":AppContainer | meta: "Button Controller" "1.0",
  state:   sleep, clock: 0, wcet: 10000, counter: 0, job: NOP, rte: none,
  comM:    "btnPort"."risingFlank" |-> "msg1",  "btnIn"."btnState" |-> "gpio63",
  memory:  bool "btnStatePrev" |-> vBool(false,0),
  inputs:  bool "btnIn"."btnState" |-> vBool(false,0)
  outputs: bool "btnPort"."risingFlank" |-> vBool(false,0) > .
```

LISTING 3.23: An application container for the *Button Controller 1.0*.

3.2.2 System Layer (CPU, Network and I/O)

Distributed embedded applications need resources and capabilities provided by the system layer in this model: They need CPU time for program execution, network communication to exchange information within and across nodes, and access to I/Os to control a technical process in the real world using sensors and actuators. This section specifies the system layer as such. It is managed by and shared with the system execution management layer described in the next section.

CPU Scheduling

Listing 3.24 shows the essential parts of the CPU scheduling specification in Maude. The CPU scheduling model heavily relies on the behavior of containers described before and thus the system layer part can be kept simple. Each *Scheduler* can assign at most one container to the CPU and set it to the *run* state. The currently running container is stored in the *current* field and the set of waiting containers is stored in the *waiting* field. A container c can be added to the scheduler's waiting set via the [enqueue] rule by sending a $sched(c)$ message. Whenever no container is running and the waiting set is not empty, an arbitrary container from the waiting set is set to *run* state by the [run] rule (which is ensured by setting $mte = 0$ in this case). When receiving a $done(c)$ message the [release] rule removes the current container c from the CPU. In essence, containers can be added to the *waiting* set and the *Scheduler* sets them to running; then the container itself observes the CPU bandwidth according to its quota configuration and sends the *done* message in time. Thus, no further logic is needed in this model to handle CPU allocation.

However, the *Scheduler* is a potential extension point towards more fine grained task scheduling during cycles if needed by future extensions (at the cost of more complex deployment modeling). For example, priorities or sub-period time slices could be introduced to achieve shorter end-to-end processing times from sensors to actuators to overcome the latencies caused by the logical execution time approach. No such violation / extension of the architectural concept is specified in this thesis, though, because the primary concern is to keep the component integration as simple as possible. For this purpose we only have two virtual priorities in this concept: The

system execution management is running with primary priority (except for background jobs), while application-layer components run with secondary priority and share the CPU time during each cycle. These priorities are not implemented using a scheduler, but by activating and running application-layer components only when the system execution manangement is done. This is done by splitting each cycle into two phases using the [enqueue] rule: The administration phase, in which the the system execution management is performed while no container is running or waiting, and the execution phase, in which no system execution management (except background jobs) are perfomed and the tasks of the application-layer components execute within containers. The execution phase is started by enqueueing the containers and the administration phase starts isochonuously after each period is over. This behavior is modeled outside the system layer in the subsequent section.

```
class Scheduler | current: Oid, waiting: Set{Qid} .
msg sched : Qid -> Msg .  --- container
rl [enqueue] : sched(C) < S:Scheduler | waiting: CS > => < S:Scheduler | waiting: C,CS > .
rl [run]: < C:Container | state: wait > < S:Scheduler | current: null, waiting: C,CS >
  => < C:Container | state: run > < S:Scheduler | current: C, waiting: CS > .
rl [release] : done(C) < S:Scheduler | current: C > => < S:Scheduler | current: null > .
eq mte(< S:Scheduler | current: null, waiting: C,CS >) = 0 .
eq mte(< S:Scheduler | >) = INF [owise] .
```

LISTING 3.24: The specification of CPU scheduling in Maude.

Network Scheduling and Routing

The system layer provides means to manage and use network communication according to the architectural concept: The primary concerns are to decouple senders and receivers, to make the temporal behavior of network communication deterministic at minimal integration effort, and to make the distributed real-time system reconfigurable during operation. To support these goals it is manageable from outside of containers whether a specific message is allowed at all, what path a message takes, and how much quota ("network bandwidth") is allocated to a message per period. Using these means the system execution management layer can (re-)configure the network communication as needed to realize a specific deployment of a distributed embedded application.

Listing 3.25 contains the essential specification of the network runtime model (cf. figure 3.7). Each network connection runtime model (*NetConR*) coordinates the transmission of network packets (*Packet*) between connected network interfaces (*NetIfR*). A network connection is an abstract representation of a shared medium used by connected network interfaces to move data between them with an appropriate protocol on top so only one network packet is transmitted at a time. The runtime model of a network connection inherits the set of connected network interfaces (*netIfs*) and the configured guaranteed bidirectional throughput (*bps*) from the design-time model (see section 3.1.3). For the purpose of transmission coordination a *NetConR* additionally holds the identifier of the currently transmitted *Packet* (if any) in the *pkt* field and for each of the network interfaces the information, whether or not they currently have packets to send. Whenever no packet is currently being transmitted and an associated interface has a packet to send with quota left, the transmission is started immediately (enforced by an *mte* equation) by the [next] rule (i.e., an arbitrary packet ready to send is picked). A *Packet* has a *value* field for the payload, *from* and *to* fields which identifiy the sending and receiving network interfaces (by node and interface identifiers), and two dynamic fields *txRestTime* and *active*. The *txRestTime* is initialized according to the size of the *Packet*

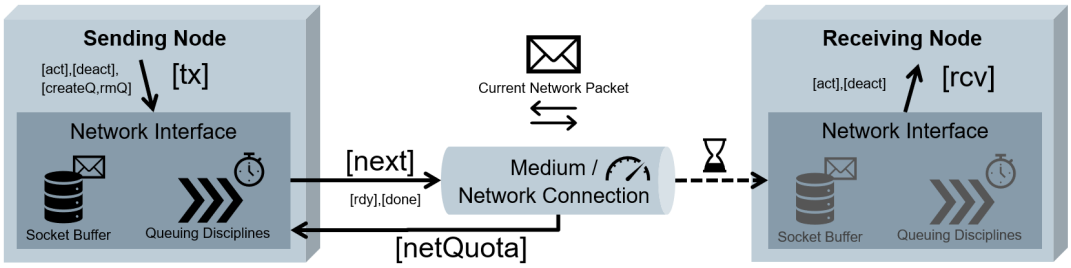

FIGURE 3.7: A schematic overview of the network runtime model. On the sending node a network packet is created via the [tx] rule. The packets are stored in a socket buffer and enqueued in the queueing discipline (Qdisc) depending on the *network message* (determining the path and quota). Transmission can be started/-continued as soon as the medium becomes free and the packet is at the head of a queue with quota left. An arbitrary Qdisc is chosen from one of the network interfaces. Tranmission is paused by the [netQuota] rule if the quota expires. Otherwise it completes as time passes, depending on the network connection's throughput and on the packet size. Finally, the [rcv] rule makes the packet contents available as *receive* message on the receiving node without involving Qdiscs and buffering.

and the troughput of the *NetConR* so that it contains the number of microseconds needed for the transmission on that medium. The *active* flag is set whenever the network packet is currently being transmitted. Whenever time passes (through the tick rule), the *txRestTime* of an active packet is adapted, so that the transmission of the packet is completed when $txRestTime = 0$. In this case, the [rcv] rule moves the network packet from the sending interface to the receiving one (potentially updating the *txReady* map through a done message, if the sending interface does not have further packets with quota to send). Alternatively, the transmission of a packet can be preempted by the [netQuota] rule, if the quota expires.

To implement this behavior, a network interface has a list of peer network interfaces, a socket buffer for storing network packets to send, and for each outgoing message a queueing discipline (*Qdisc*) for bandwidth control of associated packets. Additionally the *bps* field for the throughput and an *active* flag for whether the interface is enabled are used. The network interface runtime model also inherits the design-time model fields *port*, *ipAddr* and *macAddr*, which are not needed in this platform-independent runtime model, though. A value V can be sent as network message M via network interface I on the corresponding node using a $tx(I, M, V)$ message. In the [tx] rule a new *Packet* is then added to the socket buffer *skbuf* and enqueued to the *Qdisc* associated with network message M. A *Qdisc* has a *mesg* field, in which it stores the associated network message identifier. Additionally, a *Qdisc* has an *active* flag for whether one of its packets is currently being transmitted, a *quota* and a *used* field for the configured and the used quota, and a *queue* field for storing the identifiers of network packets to send (FIFO). Whenever one of a *Qdisc*'s network packets is currently being transmitted, it is set to active, so the *used* field is updated if time passes. If $used = quota$, then no quota is left for this network message and an *mte* equation forbids further progression of time until the transmission of the network packet was preempted. *Qdiscs* can be created and deleted on the node of the corresponding network interface I using a $setMsgQuota(I, M, N.I', S)$ message, where M is the network message identifier, $N.I'$ is the peer, and S is the "size", i.e. the quota to reserve for this network message. If $S = 0$, then the *Qdisc* is removed. Finally, network interfaces can be activated and deactivated on the corresponding

node using *activate*(I) and *deactivate*(I) messages. The [act] rule processes the activation message by resetting the *used* fields of all of the interface's *Qdiscs*, setting *active* $=$ *true* and sending a *ready* message to the network connections, for which it has network packets in its socket buffer (if any). The [deact] rule in turn sets *active* $=$ *false* for the network interface and sends *done* messages to all corresponding network connections to indicate that this interface does not want to send further network packets.

Summing it up, by configuring *Qdiscs* at network interfaces for each message both the routes and the quota of messages can be configured. This way the system execution management layer can control which messages are allowed, what path they take and how much network bandwidth they can use per cycle. This ensures that there are no hidden channels between software components and that sender/receiver relationships are defined from outside of containers. Additionally, the bandwidth control ensures that components (including system execution management components) do not take away network time from each other. The system execution management layer has to add observance of the logical execution time paradigm, so that each message is released and transmitted in a deterministic time window. For the scope of this thesis, this deterministic transmission time window is one cycle for properties or multiple cycles for larger messages (such as downloading new container images). The cycle alignment is achieved by resetting the quota and releasing messages during cycle turnover as described later in section 3.2.3. However, this network runtime model (with details such as *Qdiscs*) is also intended as a base for future work towards sub-period time slices or priority-based traffic shaping (again, for shortening the end-to-end reaction time at the cost of higher integration effort).

I/O Abstraction

Distributed embedded applications interact with a technical process using sensors and actuators. This equipment is installed on the nodes of the distributed system as modeled by the topology. As described in section 3.1.3, only GPIOs are supported at the moment. Therefore, sensors and actuators are accessible via *GpioDrivers* as modeled in listing 3.26. A *GpioDriver* has a *node* field and a *gpio* field (e.g. *node1* and *gpio13*) to identify the controlled I/O. Additionally, the *value* field models the current value, which is handled depending on the I/O type. In the abstract model presented here only two kinds of I/Os are distinguished: *GpioInputDrivers* for sensors and *GpioOutputDrivers* for actuators. The current value can be retrieved from a sensor by sending a *measure*(IO) message to the *GpioInputDriver* responsible for the corresponding *IO*. The [measure] rule answers this message with a *input*(IO, V) message containing the current value V. The [rand] rule can set a random input value at any time and is provided as an example showing how a sensor's value can be manipulated using rules: specific field behaviors and measurements (including faults) can be modeled by rules which modify the *GpioInputDrivers*' values accordingly, e.g. based on a model of the technical process and its effect on the sensor values (i.e., simulation), or based on a specific sequence of values (i.e., testing). Finally, a GPIO output can be set to the value V by sending an *output*(IO, V) message to the corresponding *GpioOutputDriver*, which is consumed by the [output] rule. Again, the effect of this output on the technical process by the behavior of the controlled actuator can be modeled by rules which modify the model of the technical process accordingly. However, the scope of this thesis ends at the I/Os, so the model of the technical process is just mentioned as a hint on how to extend this approach towards simulation scenarios (see also section 3.3).

```
class Packet | active: Bool, value: Val, from: QQid, to: QQid, txRestTime: Time.
class Qdisc | active: Bool, mesg: Qid, quota: Time, used: Time, queue: List{Qid}.
class NetIfR | active: Bool, bps: Nat, peers: Map{Qid,QQid}, qdiscs: Conf, skbuf: Conf.
class NetConR | pkt:Qid, txReady:Map{QQid,Bool}.
subclass NetIfR < NetIf. subclass NetConR < NetConn.

rl[createQ]: setMsgQuota(I,M,N.I,S) <I:NetIfR|> => <I:NetIfR| peers[M]=N.I,
  qdiscs += <Qdisc| mesg:M, active:false, quota:S+Ovhd, used:0, queue:nil> >.
rl[rmQ]: setMsgQuota(I,M,N.I,0) <I:NetIfR| peers[M]=N.I, qdiscs: QS <Q:Qdisc|mesg:M> >
  => <I:NetIfR| peers[M]=_, qdiscs:QS> .
rl[rdy]: txReady(N.I,N2.I2) <NC:NetConR| ifs: N.I, N2.I2> => <NC:NetConR| txReady[N.I]=y>.
rl[done]: noTx(N.I,N2.I2) <NC:NetConR| ifs: N.I, N2.I2> => <NC:NetConR| txReady[N.I]=y>.
rl[act]: <N:NodeR| rte: activate(I) <I:NetIfR| active:false, peers:PS, qdiscs:QS > R>
  => <N:NodeR| rte: <I:NetIfR| active:true, qdiscs: resetQuota(QS) > REST> ready?(N.I,PS,QS).
rl[deact]: <N:NodeR| rte: deactivate(I) <I:NetIfR| peers:PS, active:true> REST>
  => <N:NodeR| rte: <I:NetIfR| active:false> REST> done(N.I,PS) .
rl[tx]: tx(I,M,V) <I:NetIfR| bps:BPS, peers[M]=N2.I2, qs: QS <Q:Qdisc|mesg:M> >
  => <I:NetIfR| qs: QS <Q:Qdisc| queue+=genId(I,M)>, skbuf +=
      <genId(I,M):Packet| inactive, value:V, from:N.I, to:N2.I2, txRestTime:txTime(V,BPS)> >.
crl[next]: if T > T' then <NetConR| pkt:_, netIfs:N.I,N2.I2, txReady[N.I]=y>
  <N:NodeR|rte: R <I:NetIfR| skb:<P:Pkt|to:N2.I2>PS, qs:<Qdisc|quota:T,used:T',queue:P L> QS> >
  => <NetConR| pkt:P> <N:NodeR|rte: R <I:NetIfR| skb:<P:Pkt| act:y>PS, qs:<Qdisc|act:y> QS> >.
rl[rcv]: <NC:NetConR| pkt:P, ifs: N.I,N2.I2> <N2:NodeR| rte: <I2:NetIfR|> R2 >
  <N:NodeR|rte:R<I:NetIfR| qs:QS<Q:Qdisc|P L,mesg:M>, skb:B<P:Pkt|txrTime:0,value:V,to:N2.I2>>>
  => <NC:NetConR| pkt:_> <N2:NodeRtm| rte: rcv(I2,M,V) <I2:NetIfR|> R2> done?(N.I,N2.I2,...)
  <N:NodeR| rte: txDone(I,M,V) <I:NetIfR| qs: QS <Q:Qdisc|inactive,L>, skbuf:B> R>.
rl[netQuota]: <NC:NetConR| pkt:P> <N:NodeR| rte: R <I:NetIfR| peers:PS,
    qs: QS <Q:Qdisc| P L, mesg:M, quota:T, used:T>, skb: B <P:Pkt| txRestTime:T', to:N2.I2>> R>
  => <NC:NetConR| pkt:_> done?(N.I,N2.I2,...)
  <N:NodeR| rte: <I:NetIfR| qs: QS <Q:Qdisc|inactive>, skb: B <P:Pkt|inactive>> >.

eq mte(<NC:NetConR| pkt:_, txReady[N.I]=true, M>) = 0 .
eq mte(<I:NetIfR| qdiscs: QS <Q:Qdisc| active, quota:T, used:T', queue:P L>,
  skbuf: BUF <P:Packet| active, txRestTime: T''>>) = min(T monus T', T'') .
eq timeEffect(<I:NetIfR| qdiscs:QS, skbuf:BUF>, T)
  = <I:NetIfR| qdiscs:timeEffect(QS,T), skbuf:timeEffect(BUF,T) > .
eq timeEffect(<Q:Qdisc| active, used:T'>, T) = <Q:Qdisc| used:T'+T> .
eq timeEffect(<P:Packet| inactive, value:V>,T) = <P:Packet| value:aged(V,T)> .
eq timeEffect(<P:Packet| active, value:V, txRestTime:T'>,T)
  = <P:Packet| value:aged(V,T), txRestTime:T'-T>.
```

LISTING 3.25: The model of network scheduling and routing.

3.2.3 System Execution Management (Agent)

A real-time container agent (agent) on each node manages the system execution by (re-) configuration of and mediation between the application layer and the system layer. The agent actively controls the use of system layer resources and functionality by software components from the application layer: Which I/Os are accessible and what network messages are allowed? How much CPU time and network quota is allowed? At which instant is information exchanged between software components and their environment? These aspects must be controled by the agent to provide the required computation and communication resources while at the same time restricting them to the declared extent as of the functional and non-functional interfaces of the software components. In this way the agent can enforce the logical execution

```
class GpioDriver | node: Qid, gpio: Qid, value: Val .
classes GpioInputDriver GpioOutputDriver .
subclasses GpioInputDriver GpioOutputDriver < GpioDriver .

rl [measure]: measure(IO) <I:GpioInputDriver| gpio: IO, value: V>
  => <I:GpioInputDriver|> input(IO,new(V)) .
rl [output]: output(IO,V') <O:GpioOutputDriver| gpio: IO, value: V>
  => <O:GpioOutputDriver| value: V'> .
crl [rand]: if type(V) = type(V') then
  <I:GpioInputDriver| node: N, gpio: IO, value: V>
  => <I:GpioInputDriver| value: new(V')> [nonexec] .
```

LISTING 3.26: The abstract specification of the I/O subsystem in Maude.

time paradigm, which is the key concept for temporal decoupling of the software components. The agent utilizes the means provided by the system layer by configuring containers and network interfaces (especially queuing disciplines) according to the deployed distributed embedded application. For the timing control, the agent periodically extracts outputs, injects inputs and triggers tasks of the running software components at specific times. This together simplifies the integration (see section 3.1) and the feasibility analysis (see section 3.3) compared to priority- and timeslice-based real-time systems. This section specifies the system execution management by the agent for a fixed set of software components. The reconfiguration extensions are specified separately in section 4.1.

```
sort AgentState. ops init beforeOutputs doOutputs betweenIO doInputs afterInputs adminDone exec
      : -> AgentState [ctor] .
class Agent | node: Qid, state: AgentState, period: Nat, clock: Time, counter: Nat,
  swcs: Set{Qid}, iMap|oMap: Map{QQid,Qid}, iChk|oChk: Map{QQid,Bool},
  ifs: Set{Qid}, local: Set{Qid}, egress: Map{Qid,Qid}, ingress: Map{Qid,Qid},
  sensors|actuators: Set{Qid}, downloads: Conf, storage: Map{Prop,Val}
  rcp: RcPlan, bgm: Map{RcStep,Qid}, bgc: Qid, ES: Qid, iSync|oSync: Map{Qid,Qid}, sHops: Nat.
rl[init]: <A:Agent| init> => <A:Agent| beforeOutputs, clock:0, counter:0>.
crl[beforeOutputs]: if !mustRc(beforeOutputs,N,PLAN) then
  <A:Agent| beforeOutputs, oMap:PM, counter:N, rcp:PLAN> => mvBarriers(PM) <A:Agent|doOutputs>.
rl[sendAdm]: sendAdmin(N,V) <A:Agent| oSync[N]=M, egress[M]=I> => tx(I,M,V)<A:Agent|>.
rl[send2tx]: send(C,M,V) <A:Agent| doOutputs, egress[M]=I, oMap[C.P.PP]=M, oChk[C.P.PP]=f>
  => tx(I,M,V) <A:Agent| oChk[C.P.PP]=true>.
rl[send2out]: send(C,O,V) <A:Agent| doOutputs, oMap[C.P.PP]=O, oChk[C.P.PP]=f, actuators:O,OS>
  => output(O,V) <A:Agent| oChk[C.P.PP]=true>.
rl[send2bridge]: send(C,M,V) <A:Agent| doOutputs, oMap[C.P.PP]=M, oChk[C.P.PP]=f, local:M,MS>
  => bridge(M,V) <A:Agent|oChk[C.P.PP]=top>.
crl[oOk]:  if ok(M) then <A:Agent| doOutputs, oChk:M> => <A:Agent| betweenIO, oChk:unset(M)>.
crl[betweenIO]: if !mustRc(betweenIO,N,PLAN) then
  <A:Agent| betweenIO, sensors:IOS, inputMap:IM, counter:N, rcp:PLAN>
  => triggerSensors(IOS,IM) <A:Agent| doInputs>.
rl[rcv2swc]: rcv(I,M,V) <A:Agent|doInputs, ifs[I], ingress[M]=I, iMap[C.P.PP]=M, !iChk[C.P.PP]>
  => receive(C,P.PP,M,V) <A:Agent| iChk[C.P.PP]=true>.
rl[bridge2swc]: bridge(M,V) <A:Agent| doInputs, local:M,MS, iMap[C.P.PP]=M, iChk[C.P.PP]=f>
  => receive(C,P.PP,M,V) <A:Agent| iChk[C.P.PP]=true>.
rl[in2swc]: input(I,V) <A:Agent| doInputs, iMap[C.P.PP]=I, iChk[C.P.PP]=false>
  => receive(C,P.PP,I,V) <A:Agent| iChk[C.P.PP]=true>.
crl[iOk]: if ok(M) then <A:Agent| doInputs, iChk:M> => <A:Agent| afterInputs, iChk:unset(M)>.
crl[drop]: if !MAP[M] then rcv(I,M,V) <A:Agent| afterInputs, iSync:MAP> => <A:Agent|>.
crl[afterInputs]: if !mustRc(afterInputs,N,PLAN) then
  <A:Agent| afterInputs, counter:N, rcp:PLAN> => <A:Agent| adminDone>.
crl[setupCycle]: if T == AdminTime then <A:Agent|adminDone, clock:T, bgs:BGM, swcs:CS, ifs:IFS>
  => trigger(CS) triggerBg(BGM) activate(IFS) <A:Agent| exec>.
crl [cycleOver]: if T == P then <A:Agent| exec, clock:T, period:P, counter:N, ifs:IFS>
  => <A:Agent| beforeOutProc, clock:0, counter:N++> deactivate(IFS).
eq mte(<A:Agent| exec, clock:US, period:P>) = P monus US.
eq mte(<A:Agent| adminDone, clock:US>) = AdminTime monus US.
ceq mte(<A:Agent| S, clock:US, rcp:PLAN>) = min(AdminTime-US,mte(PLAN)) if isRcHook(S).
eq mte(<A:Agent|>) = 0 [owise].
eq timeEffect(<A:Agent| clock:US, rcp:PLAN, storage:S>,US')
  = <A:Agent| clock:US+US', rcp:timeEffect(PLAN,US'), storage:aged(S,US')>.
crl[tick]: {SYSTEM} => {timeEffect(SYSTEM,T)} in time T if T<=mte(SYSTEM) [nonexec].
```

LISTING 3.27: System execution management in Maude (without reconfiguration).

Listing 3.27 shows the compactified Maude specification of the system execution management layer without reconfiguration. The *Agent* class and the corresponding rules define the data structures and the behavior needed to manage one node (identified by the *node* field) in the distributed system according to a specific deployment. All *Agents* (one on each node) have the same *period* (equal to the period of the distributed embedded application). The *clock* field measures the time passed since starting the current cycle. The agents run their cyclic logic isochronuously in the same frequency and in the same time windows, assuming their clocks are completely synchronous. Additionally, the agent's *counter* field is incremented after each

cycle and must be synchronous for reconfiguration timing across nodes.

The agent stores the (current) configuration information in following fields. The *swcs* field holds the software components that need to be triggered on that node when starting a new cycle. The *iMap* and *oMap* fields map the local software components' properties to messages or I/Os according to the deployment description, where *iMap* contains the entries for input properties and *oMap* for output properties. The *iChk* and *oChk* map the same input and output properties to boolean flags to track whether or not a property has been sent or received in the current cycle. In addition to these application-related fields the following system-related fields are used. The *ifs* field holds references to the network interfaces on the node. The *ingress*, *egress* and *local* fields represent an abstract network configuration (i.e., firewall and routing). The *ingress* map associates allowed incoming network messages with the corresponding network interfaces. Similarly, the *egress* field maps allowed outgoing network messages to the network interfaces (thus granting and routing messages). Granted local messages are identified by the *local* set. The *sensors* and *actuators* fields refer to used GPIOs, where *sensors* are for input-providing I/Os (via *GpioInputDrivers*) and *actuators* for output-consuming I/Os (via *GpioOutputDrivers*). The following fields are primarily needed for reconfiguration purposes specified later and only mentioned here for completeness. Newly downloaded container images are stored under *downloads* and the *storage* contains captured property values. The *rcp* field contains the current reconfiguration plan and the *bgm* map tracks the currently ongoing reconfiguration steps associated with the containers they are executed in (if any). The *bgc* refers to an additional container on each node in which background operations of the system execution management are performed (e.g. downloads). For inter-agent synchronization across nodes the *iSync* and *oSync* fields map nodes to incoming and outgoing messages so that the *Agents* know how they can reach each other. This includes communication with the engineering system *ES*, an additional node from which updates are pulled. Finally, the *sHops* field holds the maximum number of hops (i.e., cycles) to other agents and is used for coordination timing.

The *Agent* manages the system execution by walking through its *AgentStates* each cycle performing the corresponding actions according to the configuration information. For now this configuration is static, i.e., it is derived from a deployment description and not modified during system execution. Before starting the first cycle the *Agent* is in the *init* state. Then both the *clock* and *counter* of the *Agent* are set to zero by the [init] rule and the *Agent* goes to the *beforeOutputs* state. This state is the starting point for each new cycle and one of three reconfiguration hooks, in which reconfiguration steps may be performed according to the current reconfiguration plan *rcp*. If the reconfiguration plan does not require any further action in this hook, the rule [beforeOutputs] starts the output processing phase. Otherwise, reconfiguration steps have to be triggered before the rule applies (which is described in section 4.1). When starting the output processing phase the *Agent* sends $getOutputCmd(C, P.PP, O)$ command messages to the corresponding containers C for each output mapping $C.P.PP \mapsto O$ in the *oMap*, where O refers to an I/O or a message. This is also referred to as 'moving the barrier' later in this thesis (hence, the *mvBarriers* operation), because in the platform-specific implementation the outputs are kept back by a barrier mechanism (see section 5). However, the platform-specific model must consider the output mapping to trigger the output extraction in the appropriate way. The *Agent* then stays in the *doOutputs* state until all expected outputs have been provided by the containers and processed by the agent. This is tracked in the *oMap* field for each property. The containers respond to the *getOutputCommand* with a $send(C, O, V)$ message as described before. Depending on the output

mapping O, one of the rules [send2tx] for network messages, [send2out] for GPIO-mapped outputs, and [send2bridge] for local messages handles the output accordingly. If O is a network message mapped to network interface I in the *egress* field, then the [send2tx] rule creates a $tx(I, O, V)$ message, so that the value V is transmitted to the receiver via network communication. If a *Qdisc* is configured for message O with enough quota for V, then the network interface will transmit the value over the corresponding connection as soon as it is activated and the medium becomes free (as described before). Similarly, the [send2out] rule hands the value V to the corresponding *GpioOutputDriver* via a $output(O, V)$ message in case of an output mapping to $O \in actuators$. As third and final possibility the [send2bridge] rule converts a $send(C, O, V)$ message to a $bridge(O, V)$ message, if the output is mapped to a local message $O \in local$. Local messages are not enqueued in a *Qdisc* in this model and can be injected to the receiving container directly in a later stage. As soon as all *oChk* flags are set, the [oOk] rule resets them to false and changes the state to *betweenIO* (between input and output processing).

This state is the second reconfiguration hook and the logical cycle turnover. After all reconfiguration operations planned for this hook are done, the [betweenIO] rule starts the input processing phase triggering all *GpioInputDrivers* for I/Os $I \in sensors$ with $measure(I)$ messages and setting the state to *doInputs*. Like outputs, inputs are processed according to three different kinds of input mappings $C.P.PP \mapsto I \in iMap$: The [rcv2swc] rule applies to network messages, the [in2swc] rule to GPIO-mapped input properties, and the [bridge2swc] rule to local messages. These two rules correspond to what is referred to as 'moving the barrier' for inputs in section 5. The [rcv2swc] rule processes network messages $rcv(I, M, V)$ received at network interface I since the last input processing phase by handing them to container C via a $receive(C.P.PP, M, V)$ message according to the input mapping, if $M \mapsto I \in ingress$. The [bridge2swc] rule converts $bridge(M, V)$ messages on that node to *receive* messages in the same way for local messages $M \in local$. And finally, the [in2swc] rule takes an $input(I, V)$ message from sensor I (sent in response to the *measure* messages) and passes the value V to the mapped container via a *receive* message according to the input mapping. Each of the three input processing rules also maintains the *iChk* field to keep track of which expected inputs have been provided. After all expected inputs are set in *iChk* the [iOk] rule ends the input processing phase resetting *iChk* and setting the *Agent* to the *afterInputs* state. This is the third and last reconfiguration hook. Besides performing planned reconfiguration operations, all non-expected input messages are actively dropped by the [drop] rule). If no further reconfiguration operation has to be performed in this hook, the [afterInputs] rule sets the agent's state to *adminDone*. While the *Agent* is in this state, an *mte* equation allows progression of (slack) time until the reserved time *AdminTime* has passed. Besides this, time can only pass during the administration phase during reconfiguration operations as described later in this chapter – all other system execution management operations are performed under ZET assumption. When $clock = AdminTime$ the [setupCycle] rule applies, which starts the execution phase of the current cycle: The state changes to *exec* and *activate* messages are sent to the containers $C \in swcs$ (and $step \mapsto C \in bgm$ in case of ongoing reconfigurations) and to the network interfaces $I \in ifs$. Another *mte* equation now allows progression of time until the period is over, i.e., until $clock = period$. During this time, the activated software components perform a cycle within their container and the network messages released during the output processing phase are transmitted. When the period is over the cycle is completed by the [cycleOver] rule, which deactivates the network interfaces, resets the *Agent*'s *clock*, increments the cycle *counter* and finally resets the *state* to *beforeOutputs*,

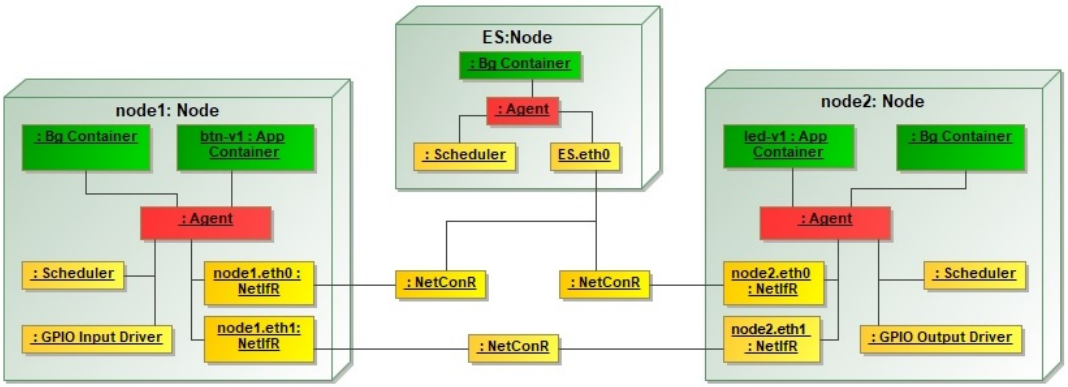

FIGURE 3.8: An UML overview of the runtime model for the *onBtnSwitch* example system. It consists of a green layer for application execution within containers, a yellow system layer for scheduling, I/Os and networking and a red layer in-between for the system execution management by an agent on each node.

which starts the next administration phase of the *Agent*.

Example 8. *Figure 3.8 shows an overiew of a runtime model instantiation for the onBtnSwitch example system. It consists of a green layer for application execution within containers, a yellow system layer for scheduling, I/Os and networking and a red layer in-between for the system execution management by an agent. The Button Controller and the LED Controller components are executed within application containers on the corresponding nodes. The additional background container is used for reconfiguration purposes, which is beyond the scope of this chapter. The agent is installed on each node and actively processes inputs and outputs of the two components during each cycle turnover in addition to triggering the cycles synchronously on each node. The engineering system node ES is connected to each node via dedicated network connections, even though it is not needed for now, as its pupose is to support rollout of dynamic reconfigurations. Figure 3.9 shows a behavioral overview of the runtime model for one cycle on node1 as timing diagram. Figure 3.10 shows a schematic timing diagram for the overall runtime model.*

In this way the *Agents* put together the application layer and the sytem layer on their nodes to run the distributed real-time system according to the logical execution time paradigm. Consequently, the timing of data processing and transmission chains is deterministic from sensors to actuators without any application-specific configuration of priorities or time-slices and thus without the need to consider special scheduling algorithms (regarding both CPU and network). The model leads to temporal decoupling of software components, which not only decreases the integration effort during design-time, but also enables the reconfiguration concept described in chapter 4.

3.3 Model Consistency

The formal specification primarily serves as a platform-independent description of the concept, but it also enables formal analysis to a certain extent: While the transition system resulting from this specification is too complex to run reachability analysis even for the simple *onBtnSwitch* example, we can still use the model to describe the consistency criteria in a precise way. We elaborate how to validate the consistency and feasibility of a distributed control system given an instance of the design-time model described in section 3.1. As one of the primary goals of this model is to

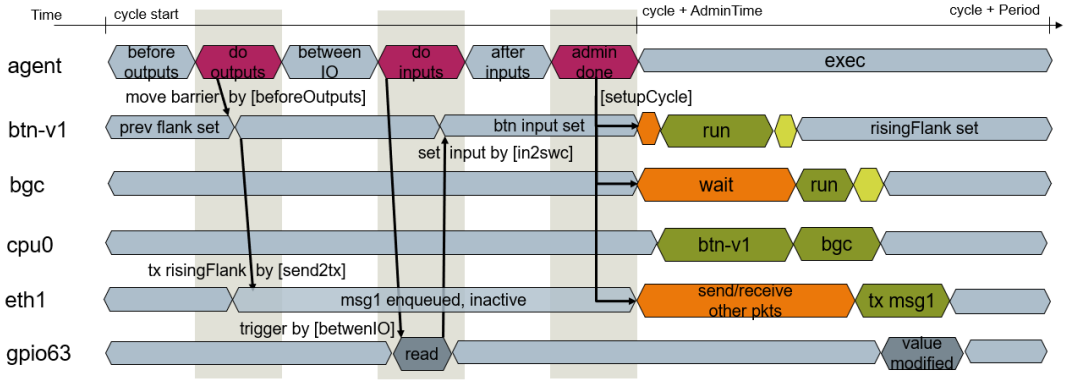

FIGURE 3.9: A behavioral overview of the runtime model at the example of one cycle on *node1* of the *onBtnSwitch* example system. The agent periodically extracts the *Button Controller*'s *risingFlank* output by moving the barrier, which is enqueued for transmission at network interface *eth1*. Then the agent injects the current state of the button from *gpio63* via the associated *GpioInputDriver*. Finally, the agent starts the execution phase by triggering the *Button Controller*, the background worker within the *BgContainer bgc*, and the network interfaces. The containers retrieve execution time by the *cpu0 Scheduler* one after another up to their configured *quota*. No reconfiguration is performed within the reconfiguration hooks or in background.

decouple engineering steps related to the different model fragments it is important to validate the fragments independently as far as possible in addition to checking the final system: We have to validate software component descriptions, distributed embedded applications, MCU descriptions, network topologies and deployment descriptions. Some consistency aspects are already ensured by the structure of the model, others have to be considered and calculated in addition. This section walks through the different aspects of consistency regarding structural and dataflow constraints as well as non-functional requirements. The consistency checks should be done as part of the engineering steps related to the corresponding model fragments and as a final overall check to ensure the consistency and feasibility of a system. For consideration and calculation we can use design-time information as well as an instantiation of the runtime model for the given deployment description. To give a coherent overview of the consistency and feasibility we also point out related properties and constraints already embodied in the meta model and described before. Consistency and feasibility regarding reconfigurations are treated in section 4.2.

3.3.1 Pre-Deployment Model Fragments

Versioned Component Types Regarding the correctness of a versioned component type the primary concern is to validate that the declared information indeed reflects the real implementation. This applies to the communication interface, the task declaration and the state interface. Each input and output of the communication interface must be declared with the correct type (determining the bit size) and timing interface (i.e., *maxAge* or *refreshmentPeriod*). The task declaration must specify an upper bound of the WCET. This is obviously essential for calculating the feasibility w.r.t. CPU time. Besides that, the requested period must conform to the needs of the control algorithm, because the algorithm might include assumptions on the control timing. The internal state declared by the task is used in this model to describe the task implementation and state transfer operations. For platform-specific models at least the size of the declared internal state must be correct to over-estimate

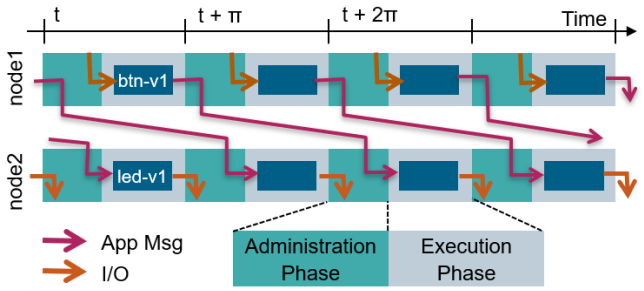

FIGURE 3.10: A schematic timing diagram of the effective behavior of the *onBtnSwitch* runtime model. The agents on both nodes synchronously perform the administration phase, which includes processing of the containers' inputs and outputs. Then they synchronously trigger the execution phase, in which the software components perform their tasks (i.e., the *ButtonController btn-v1* on *node1* and the *LED Controller led-v1* on *node2*) and in which network messages are transmitted (i.e., *msg1* from *node1* to *node2* with the *risingFlank* output of *btn-v1*).

the memory needs of the component. Finally, the state kind of the component (stateless, ground state, implicit state) must be correct, too, including the worst-case size of the persisted state (possibly in addition to the internal state) and the WCET for dumping and loading if needed. These pieces of information are used in later validation steps to ensure that a given application or deployment is feasible. However, while testing and/or checking this information is important, the scope of our model is on the platform description. As the model only uses a functional representation of the component implementation, it is too abstract to validate the interface. Thus, the component developer has to ensure that this information correctly reflects the component (e.g. by testing and profiling). Of course, a certain amount of distrust against the correctness of the component interface remains, not only because it is declared by third-party developers, but also because they are based on statistics while there may still be bugs or security issues. Therefore, instead of completely relying on testing or proof-checking components, our platform concept enforces the component interface, i.e., the architecture enforces that the component cannot reach the outside of its container in illegal ways or use more resources than declared. Beyond that it would be an interesting question for further work how we can encourage component providers to offer high-quality components along with correct component descriptions, e.g. through incentives within the business model, visible ratings or guidance and governance processes. All this is only possible in the context of industrial control systems, if we can avoid that a low-quality component can cause serious harm at runtime. This is one of the major aims of the proposed platform concept.

Distributed Embedded Applications Validation of a distributed embedded application can be done by the application designer independently from a certain deployment based on the component interfaces. Consistency aspects related to interface compatibility are already ensured by the meta model as described in the corresponding section 3.1.2: Valid connectors between the component instances' ports provide exactly one compatible matching for each required property of the ports (while provided properties may be matched by multiple connectors), where compatibility means that the types and the periods of the matched properties are equal. The *refreshmentPeriod* of the provided property and the requested period of the processing task (as of the task declaration of the component with the required property of the matching) must be equal, so that each value is processed within one cycle.

For the scope of this thesis this period must also be equal to the distributed embedded application's global cycle period, because multiple cycles per global period are not yet supported. Note that earlier in [TSK18] we proposed that multiple values should be produced and consumed within one cycle according to the quotient *period*/*refreshmentPeriod*. In this formal specification, this is instead achieved explicitly via the property's type using a structure with multiple values. If the connectors and periods are valid, then we can move on and analyze the feasibility of the distributed embedded application. Of course, the WCETs of all instantiated components' tasks must be smaller than or equal to the global cycle period of the distributed embedded application. As described in section 3.1.2, we can already make a first feasibility check regarding the worst-case reaction time of the distributed embedded application independently from the deployment based on the dataflow graph. The dataflow graph is spanned between input properties and output properties (properties as nodes). A directed edge is added between the required and the provided properties of the same component, because the component's task calculates the outputs from the inputs. Due to the LET paradigm this takes one period, which is set as these edges' weights. Additionally, for each matching defined by the connectors there is a directed edge from the provided to the required property, because the output is transmitted and provided as input. As we do not know the deployment, yet, the worst-case communication time can be any multiple of the period, including zero in case of co-located components. Thus, we can underestimate the weight of such an edge with zero for this feasibility check. Finally, there are also unmapped properties, which are going to be mapped to I/Os of the MCUs. As long as this mapping is not done, we do not know the sampling rate. The LET paradigm assumes that drivers take no time and produce/consume the inputs and outputs just during the cycle turnover, however, the period between two samples adds to the reaction time in worst-case, depending on the timing. Thus we can underestimate the additional reaction time with zero or alternatively estimate it by assuming that the sample rate is set according to the distributed embedded applications period. Consequently, the minimum possible worst-case reaction time of a distributed embedded application is $n \times period$, where n is the longest path within the dataflow graph from an input property to an output property. If this is already greater than the declared worst-case reaction time, then the distributed embedded application is not feasible at all. In this case components need to be merged to be able to do more work within one cycle, the period of which must be increased possibly. Alternatively, the application designer can check if at least the paths critical for the reaction time requirement are short enough (e.g. only check paths starting at unmapped input properties, if only reactions to real measurements are time-critical). If the application is feasible the application designer can additionally consider if $(2 + n)period$ is still within the required reaction time to check whether the I/O timing is going to be an issue (this is a platform-specific concern, though). However, this is only a first architectural feasibility check and must be resharpened later for the concrete deployment with the real sample rates and communication times.

Example 9. *The dataflow graph for the distributed embedded application of the onBtnSwitch example system is shown in figure 3.11. The longest path starts at the input property of the Button Controller, which is processed by the task within one period of time* (50 ms) *and leads to the risingFlank output property. This is then transmitted to the LED Controller (underestimating the communication time with zero). This input is processed within another period of time and results in the output property ledState, where the path ends. The path has two tasks in it, each with a fixed logical execution time of* 50 ms, *plus two additional periods*

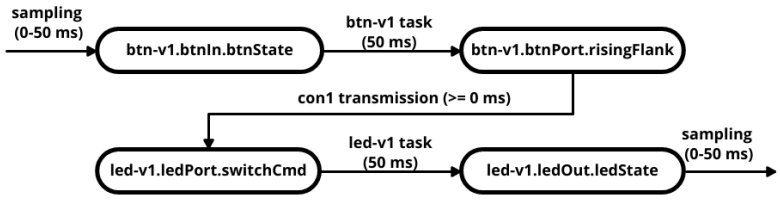

FIGURE 3.11: The dataflow graph for the distributed embedded application of *onBtnSwitch*. For processing within tasks, 100 ms are needed. Depending on the sampling rates, up to 100 ms might be added to the reaction time. Thus, if the two components are deployed to the same node, the worst-case reaction time is 200 ms.

in worst case depending on the sampling of the button input and the LED output, so the minimum assumable worst-case reaction time is $4 \times 50\,\text{ms} = 200\,\text{ms} < 250\,\text{ms} = wcrt$*, i.e., the distributed embedded application is feasible, but we only have one period margin for inter-component communication.*

Besides model consistency and feasibility, the functional correctness of a distributed embedded application is a major concern. While this is not the focus of this thesis, testing and verification of the application logic is possible based on the runtime model. For simulated tests and for model checking, the application designer does not only need the modeled distributed embedded application, but also one or more test deployments (just like at least a virtual test target would be needed for real testing). For instance, each component could be deployed to a dedicated node or all components to the same node. The testing and verification of a distributed embedded application can then be done using the runtime model as described later for the deployments.

MCU Descriptions and Network Topologies The system model is the basis for ensuring the availability of the hardware capabilities required by the distributed control system. The correctness of MCU descriptions is essential, because they specify the relevant properties of the installed hardware components (network ports, GPIOs). The key points are the throughput of the network ports and the I/O interfaces. The throughput must be a lower bound of the number of bits that can be transmitted via this hardware component bi-directionally. This model assumes that a deterministic network scheduling method is used (e.g. Time-Division Multiple Access (TDMA), Time-Sensitive Networking (TSN), ...). In this case the throughput can be taken directly from the hardware properties as no time is wasted. This work does not describe the use of such a method, though. If the real throughput is non-deterministic (e.g. due to network congestion) we proposed to reduce the declared throughput to a value the probability of which is sufficient for the purpose of the specific use case [TSK18]. Please note that this is not only intended to be an intermediate solution until determinism is introduced to the platform and the communication specification: Non-deterministic communication technologies such as TCP over WLAN do have beneficial properties for low-criticality real-time communication and should be supported side-by-side with deterministic approaches in future. The appraoch of this thesis is to take the throughput as is, instead of introducing a statistical information, because this model is much simpler to analyze. In case of non-deterministic network technologies the system architect thus has to specify a valid throughput based on long-term measurements considering the required reliability. The LET paradigm makes the system resilient against delayed network messages as long as they arrive yet within the right cycle. Within the limited scope

of this thesis we proposed (see section 3.2.3) that the agent handles the still remaining possibility of delayed network messages by checking the arrival of all expected network messages during input processing and issueing a system stop in case of a fault. Due to these considerations and given the outlook that future conceptual extensions could introduce even more resilience to the different layers and improve the fault reaction, we decided to use a hard throughput bound.

For the I/O ports of an MCU description the system architect has to provide a valid software port, i.e., which only has one interface with one property and correctly describes the data type, the direction (input or output) and the sample rate of the signal associated with the GPIO. These values are used later to check the compatibility of I/O mappings and the end-to-end reaction time. As for the component interface, the MCU interface is given to the model check as input and must be correct (garbage in garbage out). Thus, the only aspect we can check here is that each software port only has one interface with one single property. The model was designed with the goal in mind that the MCU provider can create a description of the installed hardware components together with an interface of the board support package, so that the agent can handle I/Os in a generic way based on the mapping and more complex driver logic happens in the application layer. However, the GPIOs are configured ('programmed') via the I/O port for the specific application, so the system architect must ensure that the I/O ports match the real capabilities of the hardware and the board support package, especially w.r.t. value ranges and sample rates. This can be achieved by deploying and monitoring an identification application embedded in a small physical test setup using signal generators for the inputs and oscilloscopes for inspecting the outputs.

Given valid MCU descriptions the system architect specifies the nodes and the network topology. The correctness of the modeled information is crucial, because it is used as basis for ensuring consistency and feasibility of a deployment description. It is among the goals of the topology description to enable an automatic setup of each node by the agents w.r.t. system configuration – still the system architect has to model the real topology correctly. Regarding the nodes, obviously the node identifiers must be unique, as well as their network interface descriptions' address information. It is crucial that the address information is correct (e.g. MAC addresses correspond to the real hardware). Regarding the IP addresses (as we only support IP) the addresses not only need to be unique, but also the routing must be considered. It is beyond the scope of this model to cover this aspect completely, but basically the system architect has to assign IP addresses properly so that the network interfaces connected with each other via a network connection in the topology are in the same subnet. This should be done at least as a good practice, even though in the prototype described in chapter 5 we configure routes statically to avoid discovery protocols (which also increases control) and thus would not need subnet consistency. The bidirectionally guaranteed throughput bps_{con} of each network connection *con* must be lower than or equal to the throughput declared by the corresponding network interfaces. As described in 3.1.3 this throughput must also reflect abstractions from the real network topology such as flattening (i.e., omitting switches between the nodes). Obviously, it is essential that the real network interfaces are connected as modeled apart from flattening. The system architect must also make sure that the engineering system node exists in the model and that each node is reachable by each other node (via a path through the topology graph). This task could be supported by future extensions towards topology testing, e.g. by a dedicated agent mode or by running dedicated distributed embedded applications. Note that the modeled connections

FIGURE 3.12: Graphical view of the WCET constraint: Can all tasks deployed to one node be completed within each period in addition to the administration phase and the background container?

to the engineering system not necessarily need to exist thoughout the complete system lifetime: Engineering connections are primarily used to deploy and trigger a reconfiguration plan and to perform a download (for the scope of this thesis). This can also be done by sequentially connecting to each node (which might not only be necessary in an industrial system), as long as synchronization steps are used accordingly and each node except the engineering system can continuously reach each other to send notifications and state transfer messages to each other.

3.3.2 Deployment Descriptions

Given a consistent distributed embedded application and network topology, the deployment description together with the definition of the system configuration parameters is the final integration step. We analyze consistency and feasibility of the deployment description. Additionally, we check the reactivity, i.e., whether the reaction time achieved by the dataflow from sensors to actuators fulfills the temporal requirements. Finally, we elaborate on how to use the runtime model for testing and verification.

Consistency and Feasibility Given the distributed embedded application *eApp* and the network topology *topo*, we analyze the consistency and feasibility of a deployment description, i.e., of a SWC mapping *swcMap*, a communication specification *com*, a communication mapping *comMap* and an I/O mapping *ioMap*. The SWC mapping must map each component of the distributed embedded application to a node. For each node n we then have to check the feasibility of the deployment w.r.t. CPU time (see figure 3.12). This can be calculated from the global period p of the distributed embedded application, the WCET of the administration phase e_a, the background container's quota Q_{bg} and the application containers' quota, which are equal to the WCETs of the mapped components' tasks:

$$e_a + Q_{bg} + \sum_{c \mapsto n \in swcMap} wcet(c) \leq p$$

This is basically equal to the formula proposed in [TSK18], except that meanwhile reconfiguration has been added to the concept. Reconfiguration steps are either performed directly by the agent under ZET assumption within e_a (which we have to check as described in section 4.2) or in the background container within Q_{bg}. Besides the normal cyclic administration phase, the remaining CPU is thus shared by the application containers and the background container according to their quota.

Regarding the communication specification each network message m must have an acyclic path of existing network connections from the sender node to the receiver node. The modeled messages cover both application-level and platform-level communication including communication needed during reconfiguration. For platform-level communication two default network messages with a minimum size of M_{adm}

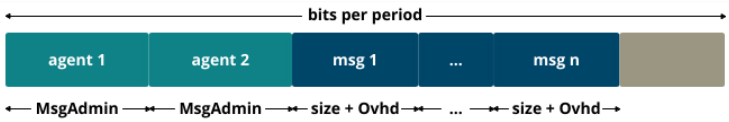

FIGURE 3.13: Graphical view of the throughput constraint: Can all messages allocated to one network connection be transmitted within a period besides the quota reserved for system execution management in both directions?

must be specified for each network connection (one for both directions) in addition to the messages needed for the connectors. The messages' bit sizes (plus protocol overhead) lead to a corresponding per-cycle quota of the throughput being allocated along the path for each message. We check the feasibility of a communication specification by ensuring that each network connection provides enough throughput for the allocated traffic given the period p and the per-message protocol overhead O_{msg}. Let M_{con} be the set of all network messages $m \in com$ the paths of which contain network connection con. Then the bidirectionally guaranteed throughput bps_{con} of network connection con is sufficient for the specified set of messages if following inequality holds (see figure 3.13):

$$\sum_{m \in M_{con}} (size(m) + O_{msg}) \leq \frac{bps_{con}}{\lceil 1\,000\,000 p \rceil}$$

On the right-hand side the throughput of the network connection is converted from bits per second to bits per period. Compared to the original proposal in [TSK18] this formula is slightly reduced on the left-hand side, because the platform-communication is now part of com. Thus in essence we only sum up the declared worst-case network traffic and compare it to the guaranteed throughput. Regarding the communication mapping we then only have to ensure that indeed each matching $c_i.p_n.pp_x \rightarrow c_j.p_m.pp_y$ defined by connectors in the distributed embedded application is assigned to an appropriate message and no message is over-used (by allocating more bits to it than its size). A message m is appropriate if its size is less than or equal to the bit size of the property's type (i.e., $size(type(c_i.p_n.pp_x)) \leq size(m)$) and if its path starts and ends at the nodes where the sending and receiving components are deployed to by $swcMap$.

The I/O mapping $ioMap$ must provide each unmapped port of the distributed embedded application with a compatible I/O port. While unused outputs could also be ignored or discarded depending on the technology (e.g. consider `/dev/null` in Linux) this stronger constraint ensures that no port is forgotten during engineering. The compatibility of I/O mappings is defined differently from the compatibility of connectors, because I/O ports are more like delegation ports (i.e., with same directions of information flow), so the idea is more to describe the API of the I/O access (in addition to instructing the agent). Thus, the software ports of the component and the I/O port must be Liskov-compatible: The types must be equal (for the scope of this thesis) and the sample rates offered by the I/O port must be at least as high as demanded. For GPIO outputs one provided property is declared with a *refreshmentPeriod*. In case of a GPIO input one required property is declared with a *maxAge*. The sample rate is sufficient if the I/O port's *maxAge* or *refreshmenPeriod* is lower than or equal to the software port's counterpart. Please note that in [TSK18] we proposed that the sample rates should be equal and that multiple values per cycle should be consumed or provided if the sample rate is higher than the task's frequency. Instead we allow the I/O interface to be faster, so we say it is valid to

get a more recent input value than required and the output value is applied faster than required. The output is thus kept on the pin as long as it is not changed by the application. However, input data available at a pin might get lost if the sample rate of the corresponding required property is to low. In this case, we propose that the solution should not be that the platform collects multiple values (as proposed earlier), but that the frequency of the task should be adapted (and thus, as of now, the period of the distributed embedded application). The reason for this decision is that collection of multiple values is only one form of input pre-processing, which should be done on application-level as it is often important for the control logic. The restriction that the global period of the distributed embedded application has to be adapted to the highest required sample rate will hopefully be overcome by future work towards sub-period cycles as enabled by Giotto [HHK03] (this approach uses task frequencies which declare how often a task must run per mode period – dynamic reconfiguration is not described, though).

Reactivity After we have checked that the model is correct and feasible we have to finally ensure that the reactivity is sufficient. Therefore we can resharpen the calculation done for the distributed embedded application based on the concrete sampling rates and communication timing resulting from the deployment description. One analytical way to check this independently from the implementation and runtime model is to take the dataflow graph (similar to the one constructed for distributed embedded applications) and enrich it with the communication and I/O timing information: The nodes of the graph are the required and provided properties (including I/O ports). For each software component there is a directed edge from all its required properties to all its provided properties. The weight is the fixed logical execution time in which the components' tasks calculate the outputs from the inputs, i.e., the period of the distributed embedded application. For each matching of each connector there is a directed edge from the provided property to the connected required property. The weight of such an edge depends on the communication mapping *comMap* and the communication specification *com*. If *com* maps the connector $c_i.p_n.pp_x \rightarrow c_j.p_m.pp_y$ to message m then the length of the *path* of message m determines the number of hops (and thus, cycles) needed for the transmission. The weight of the edge is then $p \cdot length(path)$ with period p. Note that the weight is zero for local messages, because the path is empty in this case. Finally, we add an edge for each I/O mapping entry as follows. For each mapping of an input property ${}^{\triangleright}c_i.p_n.pp_x$ to an I/O $n_k.\iota$ there is a directed edge from the qualified I/O port's property ${}^{\triangleright}n_k.\iota.pp_\iota$ to the software component's property ${}^{\triangleright}c_i.p_n.pp_x$. For each mapping of an output property ${}^{\triangleleft}c_i.p_n.pp_x$ to an I/O $n_k.\iota$ there is a directed edge in the other way, from the software component's property ${}^{\triangleleft}c_i.p_n.pp_x$ to the I/O ports' property ${}^{\triangleleft}n_k.\iota.pp_\iota$. The weights of these I/O edges can be set to the *maxAge* and *refreshmentPeriod* declared by the I/O port to analyze the reaction time between measurements and effects. Alternatively the weights of inputs edges can be set to the period p of the distributed embedded application to analyze the time between possible measurable events in the technical process and the reaction. The worst-case reaction time of the distributed control system is equal to the length of the longest path from a required I/O property to a provided I/O property within its dataflow graph. The deployment description is feasible, if this is less than or equal to the required worst-case reaction time of the distributed embedded application. It is of course a possible future direction to allow for more fine-grained reaction time constraints, e.g. requirements per pairs of inputs and outputs. However, we have postponed such possibilities and concentrated on the goal to once achieve a complete

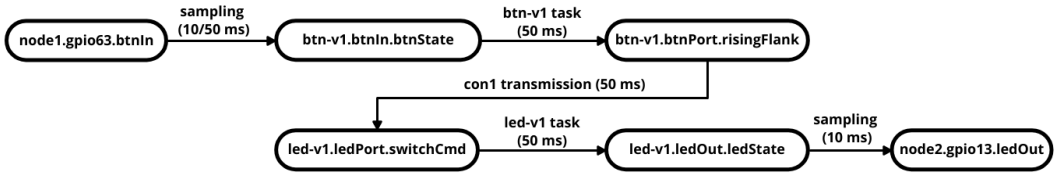

FIGURE 3.14: The dataflow graph for the deployment description of *onBtnSwitch*. For processing within tasks, 100 ms are still needed (cf. example 9). However, we now know the sample rates for the button input and the LED output as well as the communication time needed for connector *con1* (which is one period, because it is mapped to *msg1*, which takes one hop from *node1* to *node2*). Thus, the reaction time between measurement and output is 170 ms. The worst-case reaction time between measurable events and the reaction is 210 ms.

breakthrough from a decoupled design-time model down to an executable and reconfigurable distributed control system. Also note that the scope of this thesis does not cover how supervisory control or more general how access to and by external systems should be done. If everything is done by application- or platform-level components by nodes within the system scope, then of course the proposed model can be used to analyze the feasibility completely. Some aspects (such as monitoring or 'thinking') might be done via an external system in future, for which a gateway component within the system scope provides the necessary controlled connectivity. Consistency and feasibility of these aspects have to be considered by future work, because integration with external systems might result in non-functional concerns and consistency requirements beyond the current model.

Example 10. *The dataflow graph for the deployment description of the onBtnSwitch example system is shown in figure 3.14. The longest path starts at the provided property btnIn of gpio63 and goes on to the I/O-mapped input property of the Button Controller. The maxAge of* 10 ms *or the period of* 50 ms *can be set as weight, depending on the desired metric. The input is processed by the Button Controller within one period of time (*50 ms*) and leads to the risingFlank output property. This is then transmitted to the LED Controller during the next execution phase (i.e., within additional* 50 ms*). This input is processed within another period of time and results in the output property ledState, which is finally set as output within* 10 ms *according to the refreshmentPeriod declared by gpio13. The worst-case reaction time is* 170 ms *or* 210 ms *(depending on the chosen metric), i.e., the deployment fulfills the wcrt constraint of the distributed embedded application.*

Test and Verification We finally point out the possibility to test and verify the behavior of the distributed control system using the runtime model. The runtime model results from the design-time model by instantiating the derived runtime model elements with their initial states (empty queues, default values etc.). This Maude model can be used to simulate the system, so that the trace can be looked at by the engineer to get a better understanding of the dataflow timing. For instance, one cycle (50 ms) of the *onBtnSwitch* system can be simulated by running a timed rewrite of the initial configuration:

```
(trew {onBtnSwitch} in time <= 50000 .)
```

Each rule application prints a log message containing the relevant runtime information, so the engineer can see an overview of what would happen in the simulated trace. Of course, the built-in Maude features for tracing could be used, too, but the runtime model is quite big and many rules are applied during runtime simulation,

so the unfiltered output is too verbose (i.e., millions of lines for a few cycles). The most basic check enabled by the model is to check if the rewrite even reaches the given time. If it does not, then the *mte* function (which specifies the maximum time elapse allowed for the current configuration) inhibits further progression of time. The deadlocked configuration can be inspected by the engineer to identify which model element is affected and why exactly no further progress is allowed (e.g. a software component did not complete its task within the cycle, an input or output is missing or too old, an acknowledge is missing, ...). Such conditions are not reachable normally, if the design-time model was validated as described before. However, this approach is a simple alternative and can also be used to inspect reconfigurations, where such problems can occur as described later in section 4.2. The functional behavior can be tested and verified, too, even though this is beyond the scope of this thesis. A test case can be given explicitly as a sequence of values for the I/O-mapped input properties and a sequence of expected output values (or ranges). A more elaborated approach is could be specifying a simulation model of the environment in which the system is embedded (see also our reference to S# [HLR16] in section 2.3.1). Such a model adds objects and rules representing the sensors, actuators and the technical environment, which take the outputs from the *GpioOutputDrivers* of the nodes and modify the values of the *GpioInputDrivers* to simulate the interaction of the system with the technical process. Then a test case can be given by setting up situations in the simulation and monitoring the relevant conditions of the model during a simulated run of the system (i.e., running a timed rewrite and check the simulation model). In theory it would also be possible to use TCTL formulas as shown in listing 3.28 to check the reactivity or the reachability of hazards, thoug this is computationally expensive even for the small *onBtnSwitch* example system due to the complexity of our runtime platform specification.

```
--- definition of the proposition maxAgeViolation
op maxAgeViolation : -> Prop [ctor] .
eq  { <N:NodeRTM| rte: <C:AppContainer| inputs: (TY P.PP |-> V),M > RTECONF > REST }
  |= maxAgeViolation = age(V) > maxAge(C.P.PP) .

--- model checking, that the proposition cannot become true within 1s of simulation
(mc {onBtnSwitch} |=t [] ~ maxAgeViolation in time <= 1000000 .)
```

LISTING 3.28: Model checking example for whether a state is reachable within 1 sec in which any property's value becomes older than its demanded *maxAge*.

Chapter 4

Reconfiguration of Distributed Embedded Applications during Operation

Running distributed embedded applications can be reconfigured during operation using reconfiguration plans, if they comply with the base model specified in chapter 3. The base model already provides some enabling features such as the general isochronous behavior, the reconfiguration hook rules and the structures for reconfiguration plans. This makes it possible to separate the reconfiguration model into the reconfiguration extensions described in this chapter. Using these means, the complete deployment can be modified synchronously across nodes during operation – including state transfer. However, not every required reconfiguration can be achieved without quality degradation. Thus, this chapter also addresses the difficult question: How can we define and analyze consistency, feasibility and quality degradation of a distributed control system during such intrusive reconfigurations? Finally, this chapter provides reconfiguration timing templates, which use the features of the reconfiguration extensions to solve some classes of reconfiguration problems. These templates are not only illustrating examples for the reconfiguration approach, but they also give guidance on how to apply the complex reconfiguration concept at least for the cases for which we have found an abstract or general solution so far.

4.1 Model Extensions for Reconfiguration

The reconfiguration model consists of the structural definition of reconfiguration plans and a specification of the reconfiguration effect caused by such plans. Ideally, a valid reconfiguration plan would be generated from two deployment descriptions automatically to define a transition from the current deployment to the new one. While section 4.3 provides template solutions for some classes of reconfigurations, no general solution has been found so far. Therefore, reconfiguration plans are at the same time the design-time and the runtime model of reconfigurations for the scope of this thesis, i.e., an engineer has to specify reconfiguration plans for desired reconfigurations instead of only modifying the deployment description.

A reconfiguration plan is a sequence of reconfiguration steps to be executed one after another at deterministic instants on a specific node, while the system is in normal operation. Figure 4.1 shows an overview of the reconfiguration steps specified in this section. Besides steps for inter-agent coordination, most reconfiguration steps correspond to a specific part of the deployment description (component mapping, communication mapping and I/O mapping). For instance, we can add or remove a

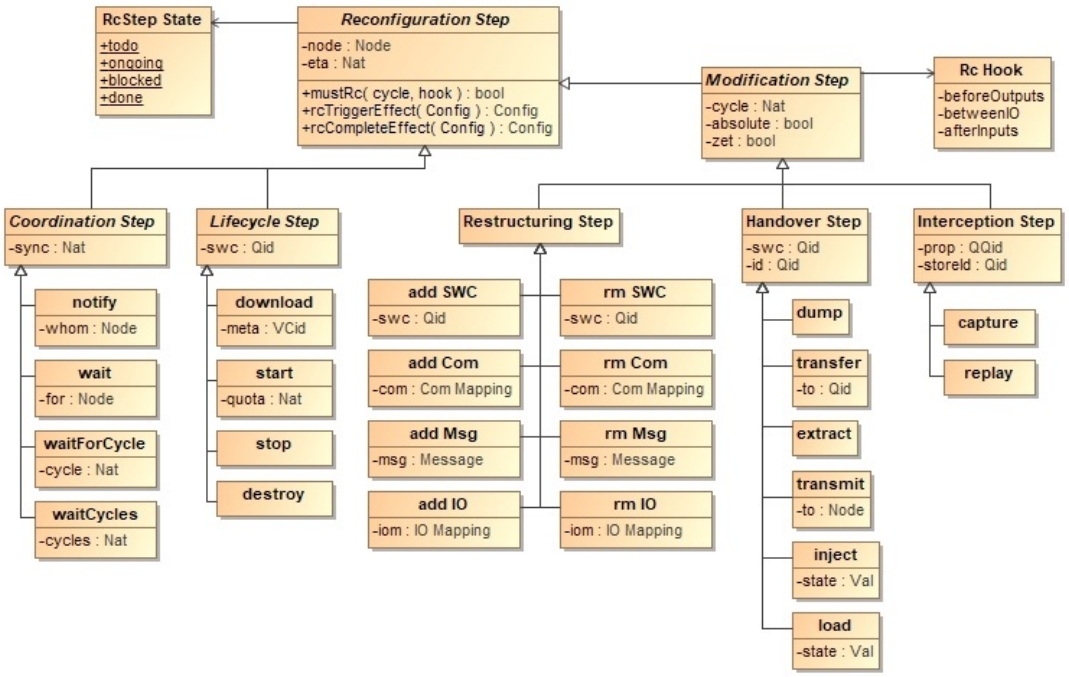

FIGURE 4.1: The defined reconfiguration step kinds in UML. There are coordination steps for inter-node coordination, lifecycle steps for container management in background, and three kinds of modification steps for time-critical modifications of a running distributed embedded application. Each kind has a specific behavior largely defined by the operations *mustRc*, *rcTriggerEffect* and *rcCompleteEffect*.

FIGURE 4.2: Overview of the basic phases of a reconfiguration. The reconfiguration is prepared in background (e.g. downloading and starting new containers). Then the involved nodes notify each other to synchronously perform the time-critical modifications (e.g. activating tasks, transferring state, ...). Finally, the reconfiguration is finalized in background (e.g. stopping old containers).

component mapping using the restructuring step *addSwc* to enable or disable triggering of the task of a component. To realize concrete reconfigurations, reconfiguration steps are deployed to each involved node as needed to transition from one deployment description to another one during full operation. Full operation means: There are no broken data processing and transmission chains, so the system maintains its integrity and reactivity. If reactivity is reduced at all, then it is only a deterministic temporal degradation, e.g. a component is blocked only for a deterministic number of cycles. This determinism is necessary for the design-time decision whether a reconfiguration is possible in the context of a given technical process (see section 4.2). To achieve this, the two primary aims of the proposed reconfiguration concept are (cf. figure 4.2):

- Prepare as much as possible in background before performing time-critical reconfiguration steps and postpone finalization work
- Avoid inter-node coordination during time-critical reconfiguration phases by performing modification steps at deterministic instants

```
ops beforeOutputs betweenIO afterInputs : -> RcHook [ctor].
ops todo ongoing blocked done : -> RcStepState [ctor].
class RcStep | node:Qid, state:RcStepState, eta:Time.
sort RcPlan < List{RcStepObj}.
op mustRc : RcHook Nat RcPlan -> Bool .
op rcTriggerEffect : RcStepObj Configuration -> Configuration [frozen (2)] .
op rcCompleteEffect : RcStepObj Configuration -> Configuration [frozen (2)] .
crl [trigger]: if mustRc(toHook(STATE),CYCLE,<S:RcStep|>) then
  <N:NodeR| rte: <A:Agent| state:STATE, counter:CYCLE, rcp: <S:RcStep|todo>PLAN > REST>
  => <N:NodeR| rte: rcTriggerEffect(<S:RcStep|>,<A:Agent|rcp:PLAN> REST)> .
```

LISTING 4.1: The base definitions for reconfiguration steps in Maude.

Listing 4.1 shows the base definitions for reconfiguration plans. The enumeration *RcHook* defines the three cyclic reconfiguration hooks (corresponding to *AgentStates*), in which reconfiguration steps can be triggered: either before output processing (*beforeOutputs*), between input and output processing (*betweenIO*), or after input processing (*afterInputs*). These hooks are essential for the timing of reconfiguration steps, because the system is 'frozen' in three different quiescent states: Before output processing all outputs have been calculated, but not been extracted from the containers. Between input and output processing, all outputs have been extracted and all input messages have been received, but not been injected, yet. After input processing, all inputs and outputs have been processed, but the tasks have not been triggered, yet. Therefore, the steps have to be performed in the right hook of the right cycle depending on the step kind and reconfiguration. All reconfiguration steps inherit from the base class *RcStep*. The enumeration *RcStepState* defines four states of a reconfiguration step from *todo* (the step has not been triggered) over *ongoing* (it has been triggered, but is not completed) and *blocked* (it is ongoing, but cannot be continued at the moment) and *done* (the reconfiguration has been completed). The other common fields of reconfiguration steps are the *node* on which the corresponding *Agent* has to trigger or execute it and the *eta* field ('estimated time of arrival'). The *eta* field is initialized with the WCET of the step and runs down as the step is executed (i.e., if time passes while $state = ongoing$) due to a *timeEffect* equation. A reconfiguration plan (*RcPlan*) is a list of *RcStep* objects. The behavior of the agent is being hooked into (cf. figure 4.3) using following operations in the style of the visitor pattern: The *mustRc* operation is used by the *Agent* in reconfiguration hook rules to find out whether the current *RcPlan* set in its *rcp* field requires immediate action. The *Agent* does not proceed with its non-reconfiguration logic as long as $mustRc(h, n, rcp)$ evaluates to *true* given the hook h (i.e., the *Agent*'s *state*), the cycle *counter* n and the current *rcp* (as of the condition of the rules [beforeOutputs], [betweenIO] and [afterInputs] defined in listing 3.27). Only the head element of the reconfiguration plan is considered. If a reconfiguration step does apply, two additional operations become relevant: The $rcTriggerEffect(step, conf)$ operation is used to trigger an applicable *RcStep* if the step is *todo*. The $rcCompleteEffect(step, conf)$ operation applies the specific reconfiguration effect of the *step* to the given model fragment *conf*, if the *step* is *done*. For most *RcSteps* the generic [trigger] rule applies the effect of triggering the reconfiguration step. The corresponding *rcTriggerEffect* equations usually mark the steps as *ongoing* and add them to the head of the *rcp*, again, leading to subsequent reconfiguration actions defined by the *rcCompleteEffect*. Which objects (including the reconfiguration plan itself) and messages are created, modified or deleted during triggering and completion of a reconfiguration step as well as when this is done depends on the kind of the *step*.

In the rest of this section, we describe the specification of each reconfiguration step kind in three groups: Coordination steps (for inter-agent synchronization),

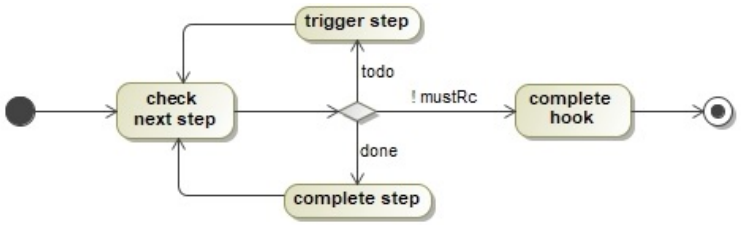

FIGURE 4.3: An overview of the reconfiguration hook activity performed by the agents on each node three times per cycle (*beforeOutputs*, *betweenIO*, *afterInputs*). The agent checks if the first step in its reconfiguration plan *rcp* requires action and performs it. If the step is *done*, the next step is considered immediately. Otherwise it blocks subsequent steps, e.g. until a corresponding response is received.

lifecycle steps (for container management) and modification steps (for time-critical modifications of running distributed embedded applications). In addition to the structure of the step kinds, the specifications contain rules and equations related to the operations *mustRc*, *rcTriggerEffect* and *rcCompleteEffect*. To improve the readability of reconfiguration plans at design-time, convenience constructor functions are defined as well for each kind of reconfiguration step. These constructure functions are also referred to as (pseudo) reconfiguration plan description language (RcPlan DSL) during this thesis. Predecessors of the RcPlan DSL were already used in [TK19a] and [TK19b]. For this formal specification, these constructor functions hide the details not needed during reconfiguration plan design, i.e., fields which can be initialized to default values (such as *todo* for the *state* field). Especially defaulting inherited fields based on the step kind can compactify the notation of reconfiguration plans. The RcPlan DSL is also used to further compactify the specification for this thesis, even when it does not reduce to matching *RcStep* objects (which is often the case when it is used on the left-hand side of equations and rules). Note that the evaluation prototype described in chapter 5 takes reconfiguration plans as input based on an implementation of this RcPlan DSL.

4.1.1 Reconfiguration Coordination

Coordination steps are used to control the timing of subsequent reconfiguration steps. Specific combinations of coordination steps can be used to achieve synchronous reconfiguration on multiple nodes, i.e., to perform reconfiguration steps temporally aligned across nodes. The coordination steps defined in listing 4.2 are (cf. synchronization steps in [TK19a]):

***notify*:** Send a notification with timing information to another node.

***wait*:** Wait for a notification from another node.

***waitForCycle*:** Wait until a specific absolute cycle (e.g. start-up sequences).

***waitCycles*:** Wait for a given number of cycles (e.g. intermediary waiting periods).

The latter two coordination steps are single-node steps to delay reconfiguration plan execution to a later cycle. The reconfiguration step $waitForCycle(c, k)$ blocks subsequent reconfiguration steps until cycle c is reached as of the *Agent*'s *counter*. In cycle c the *mustRc* equation evaluates to *true*, so the *rcTiggerEffect* equation for *WaitCycleStep* is applied to the *rte* configuration on the corresponding node. This only modifies the *rcp* of the *Agent* using the *planSetAbsCycles*(c,*rcp*) operation, which changes the timing information of modification steps as described later, so that they are now aligned to the current cycle c. When a reconfiguration step $waitCycles(c', k)$ becomes

```
class CoStep | sync:Nat.
class NotifyStep | whom:Qid.              class WaitCycleStep | cycle:Nat.
class WaitStep | for:Qid.                 class WaitCyclesStep | cycles:Nat.
subclasses NotifyStep WaitStep WaitCycleStep WaitCyclesStep < CoStep < RcStep .

eq notify(N,K) = <NotifyStep| todo, node:_, eta:0, whom:N, sync:K>.
eq wait(N,K) = <WaitStep| todo, node:_, eta:0, for:N, sync:K>.
eq waitForCycle(C,K) = <WaitCycleStep| todo, node:_, eta:0, cycle:C, sync:K>.
eq waitCycles(C,K) = <WaitCyclesStep| todo, node:_, eta:0, cycles:C, sync:K>.

eq mustRc(beforeOutputs,C,<S:NotifyStep|> P) = true.
eq mustRc(H,C,<S:WaitCyclesStep|> P) = true.
eq mustRc(H,C,<S:WaitCycleStep| cycle:CC> P) = C >= CC.

eq rcTriggerEffect(<S:NotifyStep| node:N, whom:N2, sync:K>,
  <A:Agent| node:N,counter:C, rcp:P, oSync[N2]=M, egress[M]=I, sHops:H> REST)
  = tx(I,M,vNfy(N,K,cc(C,H),size(vNfy)) <A:Agent| rcp:waitForCycle(cc(C,H),K) P> REST.
rl[rcvNfy]: <A:Agent| afterInputs, ifs:I,IFS, iSync[M]=N', rcp:wait(N',K) P> REST
  rcv(I,M,vNfy(N2,K,C,S)) => <A:Agent| rcp:waitForCycle(C,K) P> REST.
eq rcTriggerEffect(<S:WaitCyclesStep| node:N, cycles:0, sync:K>, <A:Agent| counter:C, rcp:P> R)
  = <A:Agent| rcp:waitForCycle(C+0,K) P> R.
eq rcTriggerEffect(<S:WaitCycleStep|>, <A:Agent| counter:N, rcp:P> REST)
  = <A:Agent| rcp:planSetAbsCycles(N,P)> REST.
```

LISTING 4.2: The definition of coordination steps in Maude.

the head of the reconfiguration plan during a hook in cycle c, the corresponding *rcTriggerEffect* equation immediately (*mustRc* is always *true*) replaces the step with a $waitForCycle(c + c', k)$ step, so subsequent reconfiguration steps are not performed until cycle $c + c'$ is reached. Thus, the *cycle* field of *WaitCycleSteps* is the absolute continuation cycle while the *cycles* field of *WaitCyclesSteps* is relative. The absolute variant is used by the relative one, but also by *notify* and *wait*.

The multi-node coordination steps *notify* and *wait* determine a common continuation cycle *cc* on two or more nodes. The mechanism uses the fact that all *Agents* run the cycles isochronously with fully aligned *clocks* and cycle *counters* and that $sHops \times period$ is an upper bound of the worst-case communication time between *Agents*. The basic synchronization protocol is arranged by deploying the step $notify(n_2, k)$ on node n_1 and $wait(n_1, k)$ on node n_2. Due to the first *mustRc* equation, a *NotifyStep* is triggered as soon as the agent reaches the reconfiguration hook *beforeOutputs*, i.e., at the start of the next (or the current) administration phase, before outputs are released from the containers. This hook is fixed to keep the timing of this platform-independent concept independent from whether the concrete platform implementation keeps back all outgoing network messages (including inter-agent communication) released after the output processing phase. The correponding *rcTriggerEffect* calculates the common continuation cycle $cc = counter + sHops + 1$ and waits for it by adding $waitForCycle(cc, k)$ to the head of the reconfiguration plan. Additionally, *cc* and the synchronization point identifier k are sent to n_2 via the network message m according to the outgoing synchronization mapping (i.e., $n_2 \mapsto m \in oSync$). Inter-agent messages are transmitted over the mapped network interface i during execution phases like properties (i.e., bandwidth controlled and hop-by-hop), so the notification is received by n_2 before the continuation cycle *cc*, if *sHops* is greater than or equal to the number of hops between n_1 and n_2 on the path of m. Unlike most reconfiguration steps, the $wait(n_1, k)$ step is triggered on n_2 by a dedicated rule [rcvNfy] because the trigger condition is not only a specific cycle and hook, but receiving a notification message for synchronization point k. The rule replaces the *WaitStep* with a $waitForCycle(cc, k)$ according to the content of the received notification. In this model it is important that the notification information is still available, in case the *Agent* on n_2 has received it before reaching the *wait* step in its reconfiguration plan. The notification is processed in the reconfiguration

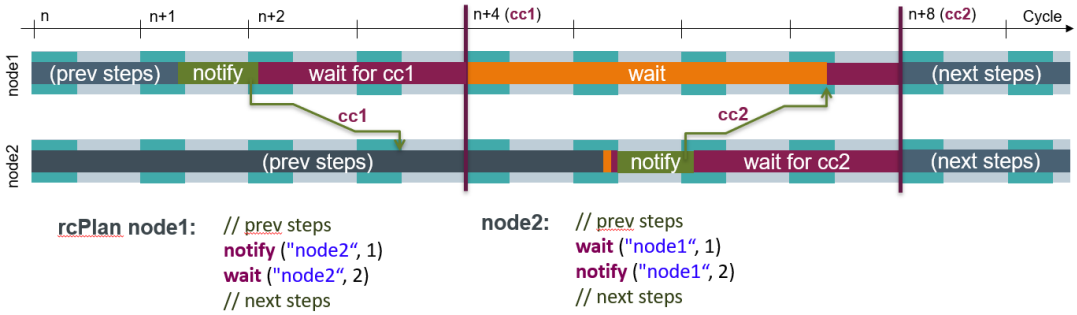

FIGURE 4.4: A schematic timing diagram of a possible run of the two-way handshake. The reconfiguration step at the head of the plan is shown for both nodes over time. After *node1* completes the previous steps, it reaches the *notify* step. In the next hook *beforeOutputs* it sends the continuation cycle number *cc1* to *node2* and waits for it. As *node2* is still busy it stores this information and only process it after the previous steps in cycle $t + 5p$ (after *cc1*). Thus, *wait* is immediately replaced by *waitForCycle(cc1)*, which immediately completes, so the *notify* step becomes the next step. The second continuation cycle number *cc2* is sent to *node1*, which already waits for the notification since *cc1*. Then both nodes wait for *cc2* and then continue synchronously with subsequent reconfiguration steps.

hook *afterInputs*, i.e., at the end of the administration phase, after the input messages were processed. Again, this hook is chosen to keep the protocol deterministic in case platform-specific implementations only release incoming network messages (including inter-agent communication) during input processing. The result of this basic synchronization protocol is that none of both nodes will continue before *cc*, especially that n_2 does not continue before n_1: When n_2 reaches the *wait* step in cycle $c < cc$ before the notification arrives, both nodes will continue synchronously in the continuation cycle *cc* (because both use *waitForCycle*(*cc*,k)). Otherwise, when n_2 reaches the wait step in a later cycle $c >= cc$, it continues immediately, but n_1 has already continued earlier in cycle *cc*.

This basic synchronization protocol can be used to let multiple nodes continue synchronously. For two nodes a two-way handshake is needed (see figure 4.4): Node n_1 performs $[notify(n_2, k), wait(n_2, k')]$ and n_2 performs $[wait(n_1, k), notify(n_1, k')]$. Using such a two-way handshake it is guaranteed that both nodes continuously proceed with subsequent reconfiguration steps: None of the nodes will continue with their second steps before reaching the continuation cycle of the first synchronization point k, so n_2 sends the notification for the continuation cycle of the second synchronization point k' while n_1 already awaits it (which, as described before, leads to synchronous continuation). For more than two nodes this handshake can be extended by letting one node n_m wait for notifications from all involved nodes and then notify them in turn (a two-phase multi-way handshake). For the multi-way handshake, a multicast notification is needed on n_m, which does not wait between the notifications in the second phase but sends them all in the same cycle. Finally, each coordination step is associated with a synchronization point identifier k. For *notify* and *wait* this is used to identify synchronization points across nodes. The main intention of the synchronization points is to enable future extensions towards concurrent reconfigurations, which might associate modification steps with synchronization points (currently, only the most recent synchronization point is used for timing as described in section 4.1.3).

4.1.2 Container Lifecycle Management

Each software component instance is executed in its own container. During dynamic reconfigurations, the containers need to be managed therefore, which is done by the lifecycle steps specified in this section. Following lifecycle steps are defined in listing 4.3 (cf. background steps in [TK19a]):

***download*:** Create a new component instance from a container image.
***start*:** Start a container to prepare a new component for activation.
***stop*:** Stop a container after the component has been deactivated.
***destroy*:** Destroy a stopped container.

```
class LcStep | swc:Qid.
class DownloadStep | meta:VCid.      class StartStep | quota:Nat.
class DestroyStep.                   class StopStep.
subclasses DownloadStep StartStep StopStep DestroyStep < LcStep < RcStep .

class DlReq | meta:VCid.             class Image | meta:VCid, proto:SwcObject.
subclasses DlReq Image < Serializable | size:NzNat.

eq download(V,C) = <"dl-{C}":DownloadStep| todo, node:_, eta:rand(), swc:C, meta:V>.
eq start(C,T) = <"start-{C}":StartStep| todo, node:_, eta:rand(), swc:C, quota:T>.
eq stop(C) = <"stop-{C}":StopStep| todo, node:_, eta:rand(), swc:C>.
eq destroy(C) = <"destroy-{C}":DestroyStep| todo, node:_, eta:rand(), swc:C>.

eq mustRc(betweenIO,N, <S:LcStep|todo> REST) = true .
eq rcTriggerEffect(download(V,C), <A:Agent| ES:E, oSync[E]=M, egress[M]=I, bgc:B, rcp:P> REST)
 = tx(I,M,<DlReq|meta:V,size:MA>) <A:Agent| bgm[<S|>]=B, rcp:<S|ongoing> P> REST.
eq rcTriggerEffect(start(C,0), <A:Agent| dl:<C:SWC|>D ,bgc:B, rcp:P> REST)
  = new(<C:SWC|>) job(B,<S:StartStep|>) <A:Agent| bgm[<S|>]=B, rcp:<S|ongoing> P> REST.
eq rcTriggerEffect(<S:LcStep|>, <A:Agent| bgc:B, rcp:P> REST)
  = job(B,<S:LcStep|>) <A:Agent| bgm[<S|>]=B, rcp:<S:LcStep|ongoing> P> REST [owise].
rl [lcDoneMsg]: <N:NodeR| jobDone(<S:LcStep|>) <A:Agent| rcp:<S:LcStep|> P> REST>
  => <N:NodeR| rcCompleteEffect(<S:LcStep|>, <A:Agent| rcp:<S:LcStep|done> P> REST)>.
rl [lcComplete]: <A:Agent| betweenIO, bgm[<S:LcStep|>]=B, rcp:<S:LcStep|done> P>
  => <A:Agent| rcp:P, rm(bgm,<S|>)>.
rl [txImage]: rcv(I,M,<DownloadRequest| meta:V>) <A:Agent| iSync[N]=M, oSync[N]=M2,
  egress[M2]=I2, dl:<Image|meta:V> REST> => <A:Agent|> tx(I2,N,M2,<Image|>)).
rl [rcv2dl]: rcv(I,M,<Image|meta:V,proto:<SWC|>) <A:Agent| rcp:download(V,C) P,  dl:REST>
  => <A:Agent| rcp:<DownloadStep| done> P, dl:clone(C,<SWC|>) REST>.
eq rcCompleteEffect(start(C,0), <C:AppContainer| new> REST) = <C:AppContainer| sleep> REST.
eq rcCompleteEffect(stop(C), <C:AppContainer| sleep> REST) = <C:AppContainer| stop> REST.
eq rcCompleteEffect(destroy(C), <C:AppContainer| stop> REST) = REST .
```

LISTING 4.3: The definition of lifecycle steps in Maude.

The four lifecycle steps are derived from *LcStep*, which has the instance identifier *swc* of the corresponding software component in addition to the base attributes of the root base class *RcStep*. The lifecycle of a container c is normally controlled using the steps $download(meta, c)$, $start(c, quota)$, $stop(c)$ and $destroy(c)$ (in this order). These steps are always executed in background, i.e., the steps are triggered during the reconfiguration hooks, but their execution is done while the system continues to operate. All reconfiguration steps (except *download*, *dump*, and *load*) which need to be performed in background are executed in an additional, non-application container of type *BgContainer*, which is specified in listing 4.4 (see figure 4.5). A *BgContainer* is a normal *Container* with slightly different behavior. A *Container* has a *job* field, which is only used for reconfiguration steps. Via the [bgNew] rule a $job(c, step)$ message can be used to trigger the execution of a reconfiguration step within the *Container* c. The step is set to *ongoing* and due to the *timeEffect* equation its *eta* runs down as time passes when the *Container* is in *run* state (i.e., running on the CPU). When the *eta* has run down to zero, the [bgDone] rule immediately removes the *job*, sends a $jobDone(step)$ message and goes to the *over* state to release the CPU. If the *Container*'s *quota* expires before the step is completed it is preempted by the [quota]

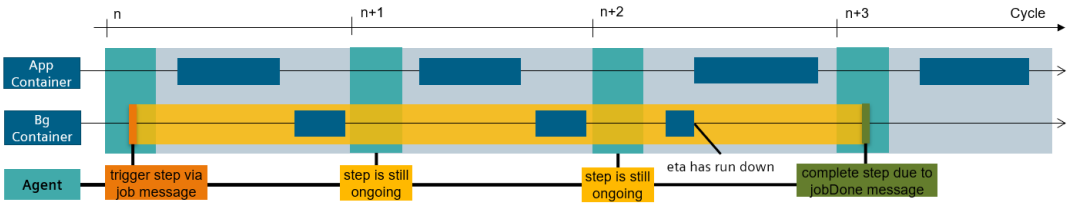

FIGURE 4.5: A schematic timing diagram of a lifecycle step performed in background. The agent triggers lifecycle steps by sending a *job* message to the *BgContainer*. Then the reconfiguration step's *eta* runs down whenever the *BgContainer* gets CPU time according to its *quota*. The step stays at the head of the agent's list and blocks subsequent steps until the *BgContainer* answers with a *jobDone* message. Meanwhile, the *AppContainers* are run and managed normally.

rule and continued in the next cycle. This basic behavior of *Containers* also enables execution of the specific *RcSteps dump* and *load* in *AppContainers* with slightly different completion behavior as described later. However, the *jobDone* message indicates completion of an *LcStep* in the *BgContainer* and is consumed by the *Agent* in the [lcDoneMsg] rule. Triggering and checking the status of lifecycle steps is done once per cycle in the hook *betweenIO* as of the *mustRc* equation. Only after the step is completed and the reconfiguration effect was applied using the *rcCompleteEffect* operation or step-specific rules, subsequent steps will be considered (e.g. *CoSteps* to create a synchronization point).

```
class BgContainer < Container.
rl [bgNop]: <B:BgContainer| run, job:_> => <B:BgContainer| over>.
rl [bgNew]: job(C, <S:RcStep| todo>) <C:Container|> => <C| job:<S:RcStep| ongoing> >.
rl [bgDone]: <B:BgContainer| job:<S:RcStep| ongoing, eta:0> >
  => <B:BgContainer| over, job:_> jobDone(<S:RcStep|>).
eq mte(<B:BgContainer| run, job:_>) = 0 .
eq timeEffect(<C:Container| run, job:<S:RcStep| ongoing, eta:N>, clock:T>, dT)
  = <C:Container| job:<S:RcStep| eta:N-dT>, clock:T+dT>.
```

LISTING 4.4: Maude specification for performing *RcSteps* as background jobs.

A new container is downloaded and instantiated using a $download(meta, c)$ step, where the *meta* information (i.e., the versioned component identifier) needs to be given in addition to the instance identifier c. The *rcTriggerEffect* equation for *DownloadSteps* sets the step to ongoing and sends a download request message *DlReq* containing the *meta* information to the engineering system *ES* via the configured inter-agent message. Only *Serializable* objects can be transmitted over network, because the *size* information is needed for network scheduling. The *size* of *DlReq* objects is over-estimated with the agent's quota M_{adm} and thus needs one cycle for transmission to *ES*. The agent on the engineering system answers the request by sending the corresponding container image at the configured quota as specified by the [txImage] rule. The *Image* also contains the *meta* information, which is equal to the *meta* information from the download request. Additionally, the *proto* information is contained, which holds a prototype object of the software component model, i.e., an *SWC* object describing the component interface (see section 3.1.1). The *size* field inherited from the *Serializable* class specifies the bit size of the *Image* and also determines the transmission duration. When the software component is received in an *Image*, the [rcv2dl] rule stores it under *downloads* setting the instance identifier c and marking the *DownloadStep* as *done*. The step then blocks subsequent *RcSteps* of the plan until the [lcComplete] rule removes it from the plan and from the map of ongoing background steps in the next *betweenIO* hook (depending on the platform-specific model it would be a valid alternative that the *jobDone* message is only consumed

betweenIO). This way the container image is pulled from the component repository in background. For example, $download(LED\ Controller^{1.0}, led_1^{1.0})$ creates a new software component instance $led_1^{1.0}$ on the corresponding node based on the container image $LED\ Controller^{1.0}$. The duration of a *DownloadStep* is determined basically by the *Image*'s *size* and the configured *quota* reserved for inter-agent communication between the node and the engineering system: The request is sent during one cycle and answered with the image in $\lceil size/quota \rceil$ more cycles. However, this calculation is only informational and the reconfiguration plan should use *CoSteps* after lifecycle steps to synchronize reconfigurations across nodes, if needed (e.g. consider future extensions towards downloading from an external source).

The downloaded software component can be started in a new *AppContainer* c in background using a $start(c, quota)$ step, where the allocated CPU bandwidth can be modified using the *quota* attribute. The *rcTriggerEffect* for *StartSteps* sends a *job* message to the background container and creates the *AppContainer* instance from the downloaded meta information. The container is set to state *new* with initial property valuations for *inputs*, *outputs* and *memory* according to the component interface. The communication mapping *comM* is left empty so the component is completely isolated in the container, initially. After receiving the *jobDone* message from the background container, the *rcCompleteEffect* is applied immediately by the [lcDoneMsg], which changes the state of the *AppContainer* from *new* to *sleep* and marks the step as *done*. The step is removed by the [lcComplete] rule in the next *betweenIO* hook (modeling that the agent notices the changed container state at that time). If zero is given as *quota*, the CPU bandwidth is left defaulted to the software compontent's WCET taken from the downloaded *Image*'s meta information. For instance, $start(led_1^{1.0}, 0)$ starts the container of $led_1^{1.0}$ in background configuring a CPU bandwidth of e/p, if $e \in \mathbb{N}$ is the WCET of the task of $led_1^{1.0}$ and $p \in \mathbb{N}$ is the period of the running distributed embedded application. A non-zero *quota* can be given, which sets the bandwidth accordingly. Tasks running in such an application container may take multiple cycles until completion if $quota < wcet$. This feature is needed for update package components as described later in chapter 4.3. It is important to note that starting a container does not yet impact the running distributed embedded application: Though the container is running, the task of the component is not triggered and inputs and outputs are not processed without subsequent reconfiguration steps described in the next section. The containerization of both the reconfiguration step and the component ensures that no side effects besides initialization and resource allocation are possible, until further reconfiguration steps enable them. A main requirement for platform implementations is that this isolation must be enforced by the containerization independently from whether software component implementations adhere to their declared interfaces, therefore.

Finally, containers are stopped and destroyed given the component identifier c in the similar way using a generic *rcTriggerEffect* rule for *LcSteps*, which sends the step to the background container as a *job* message and marks it as ongoing. After the background container has received enough CPU time according to its quota so that the *RcStep*'s *eta* has run down to zero, the *jobDone* message is issued and leads to the corresponding *rcCompleteEffect* due to the [lcDoneMsg] rule. A $stop(c)$ step has the reconfiguration effect that the container changes its state from *sleep* to *stop*. A $destroy(c)$ step has the effect that the *AppContainer* c is removed from the node.

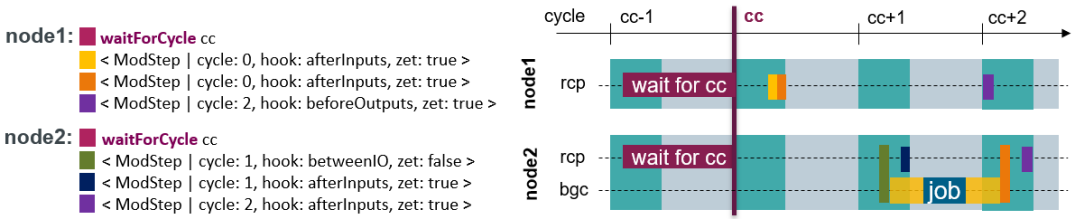

FIGURE 4.6: A schematic timing diagram of how modification steps are triggered and executed. On the left-hand side, the reconfiguration plans for both nodes are shown. Both nodes wait for the same cycle *cc* (e.g. after performing the synchronization protocol). On *node1* three modification steps are performed under ZET assumption, i.e., directly within the reconfiguration hooks in the specified relative cycles. On *node2* the first step is performed in background within a container. It does not block the subsequent steps even while it is still ongoing.

4.1.3 Transitioning between Applications

Modification steps are reconfiguration steps which modify the running system to effectively transition from the deployed distributed embedded application to another one. Three different groups of modification steps are specified in this section: restructuring steps, handover steps and interception steps. They have essential commonalities abstracted in the base class *ModStep* (see listing 4.5). In earlier work, *ModSteps* were referred to as real-time steps, bounded background steps, and bounded in-component steps, depending on how they were triggered and executed [TK19a; TK19b]. The different modification steps are now grouped by their reconfiguration purpose, because they are more common and also because most steps can be triggered and executed in multiple anyways. For instance, in [TK19b] there were two versions of extracting state from a container – the real-time step *extract* and the bounded background step *extractBg*. Now there is only one kind of handover step for extraction, the execution mode of which is configurable like for most other *ModSteps*. Therefore, we decided that it is clearer to use reconfiguration commonalities for grouping rather than operational ones (which also lead to duplicates). In the example of the *extract* step, its commonality with other handover steps is that they refer to the state of a software component (see later in this section).

```
class ModStep | cycle:Nat, abs:Bool, hook:RcHook, zet:Bool.
subclass ModStep < RcStep.

eq mustRc(H,N,<S:ModStep| hook:H, cycle:N, abs:true> P) = true.
eq rcTriggerEffect(<S:ModStep| zet:true>, <A:Agent| rcp:P> REST)
  = <A:Agent| rcp:<S:ModStep|ongoing> P> REST.
eq rcTriggerEffect(<S:ModStep| zet:false>, <A:Agent| bgc:B> REST)
  = job(B,<S:ModStep|>) <A:Agent| bgm[<S:ModStep|>]=B> REST [owise].
rl [zetDone]: <N:NodeR| <A:Agent| rcp:<S:ModStep| ongoing, eta:0, zet:true> P>  REST>
  => <N:NodeR| rcCompleteEffect(<S:ModStep|>, <A:Agent| rcp:P> REST)>.
rl [modBgDoneMsg]: <N:NodeR| jobDone(<S:ModStep|>) <A:Agent| bgm[<S:ModStep]=B> REST>
  => <N:NodeR| rcCompleteEffect(<S:ModStep|>, <A:Agent| rm(bgm,<S|>) > REST)>.

eq mte(<S:ModStep| ongoing, zet:true, eta:T> P) = T.
eq timeEffect(<S:ModStep| ongoing, zet:true, eta:T> P, T') = <S:ModStep| eta:T-T'> P.
```

LISTING 4.5: General definitions for modification steps in Maude.

ModSteps need to be performed in strictly deterministic temporal alignment relative to each other – possibly across nodes. Only then we can maintain data processing and transmission chains throughout the reconfiguration. To make this possible, the timing of triggering modification steps is determined from their fields as follows (see figure 4.6): First, the *cycle* field is given, which is interpreted either as absolute or relative cycle number depending on *abs*. If *abs* is *true*, then the step is

triggered in the reconfiguration hook *hook* during the administration phase of the given *cycle*. This is achieved by comparing the *cycle* with the given *counter* value of the *Agent* in the *mustRc* equation. Else, if *abs* is *false*, then the *cycle* is interpreted relative to the most recent synchronization point resulting from previous coordination steps. To achieve this behavior the relative *cycles* are converted to absolute ones by *WaitCycleSteps* (which are implicitly used by all *CoSteps* as described before). For instance, a modification step with $abs = false$, $cycle = m$ and $hook = betweenIO$ after a synchronization step $waitForCycle(n)$ is triggered in the reconfiguration hook *betweenIO* of the absolute cycle $n + m$ (m cycles after the most recent synchronization point's continuation cycle n). Besides the timing of the triggering, it is essential whether the modification step can be executed directly in the reconfiguration hook under ZET assumption or the step must be executed in background. Therefore, two *rcTriggerEffect* equations are defined in listing 4.5 depending on the *zet* field. If *zet* is *true*, then the step is set to ongoing and prepended to the plan again. The *timeEffect* counts down the *eta* field of the step as time passes until $eta = 0$. Then the reconfiguration effect is applied still within the hook by the generic [zetDone] rule using the *rcCompleteEffect* operation, which is defined differently for each kind of *ModStep*. Thus, in the ZET case the modification step is executed and removed 'immediately' (within the reconfiguration hook), while the distributed embedded application is quiescent. Instead, if *zet* is *false* then the step is still triggered at the same time, but it is not executed and completed within that reconfiguration hook. In this case, the second *rcTriggerEffect* equation matches, so the agent sends a *job* message to the specific container, which has to execute the modification step in background. For most steps, this is the *BgContainer* running on each node, referred to by the agent's *bgc* field (which is also used for lifecycle steps). Only two steps need to be performed (and implemented, cf. section 3.1.1) by software components within their *AppContainers*, i.e., dumping and loading state in an application-specific data format. The other exception only mentioned here is the *transmit* step, which is 'backgrounded' using the bandwidth control mechanism of the network specification. However, when triggering a *ModStep* for execution in background, the agent moves the step from the reconfiguration plan to the *bgm* map of pending reconfiguration steps (mapping it to the responsible container). The container reports completion of the step (i.e., that *eta* has run down to zero) via a *jobDone* message as described before (cf. listing 4.4 and figure 4.5). The agent processes the *jobDone* message by applying the step-specific *rcCompleteEffect* and removing the step from the *bgm* map as specified in the [modBgDoneMsg] rule. In contrast to lifecycle steps, modification steps do not block subsequent reconfiguration steps while they are running in background. Further modifications can be performed instantly or in later hooks and cycles, even if a modification step is still *ongoing* or *blocked* on the same node. The number of cycles needed until a backgrounded *ModStep* is completed is deterministic given its *eta* and the responsible container's *quota*: The step is completed within $n = \lceil eta/quota \rceil$ cycles after triggering (if not, it is an error condition). The reconfiguration effect is available at this point, e.g. after triggering a state transfer in cycle c_0 the state can be loaded by a subsequent modification step in $cycle = c_0 + n$. Obviously, this calculation of the temporal bounds applies only if steps are not triggered in parallel within the same container. It is a design-time responsibility to check the feasibility of the reconfiguration step timing and the consistency of the reconfiguration effects (see section 4.2). In the remaining section, the three kinds of modification steps are specified - restructuring steps, handover steps, and interception steps.

Restructuring Steps

Restructuring steps change the structure of a running distributed embedded application. The structure is basically given by a distributed embedded application's deployment, though each *Agent* has its own node-local view. Especially during reconfigurations the aggregate of these views does not always yield a valid deployment. Following restructuring steps are defined to modify the SWC mapping, the communication mapping, and the I/O mapping during operation (see listing 4.6):

***addSwc/rmSwc*:** modify the SWC mapping (start or stop triggering a task).
***addMsg/rmMsg*:** modify the communication specification (add/remove a message).
***addCom/rmCom*:** modify the communication mapping (add/remove a matching).
***addIO/rmIO*:** modify the I/O mapping (enable/disable I/O-mapped property).

```
class RsStep.                          ops abs rel _: Nat -> CycleSpec [ctor].
ops in out: -> Direction [ctor].       ops local ingress egress: -> MsgKind [ctor].
classes AddSwcStep RmSwcStep | swc: Qid.
classes AddComStep RmComStep | dir: Direction, prop: QQid, mesg: Qid.
classes AddMsgStep RmMsgStep | m: Qid, kind: MsgKind, size: Nat, route: Qid, peer: QQid.
classes AddIomStep RmIomStep | dir: Direction, prop: QQid, io: Qid.
subclasses AddSwcStep  RmSwcStep  [...] RmIomStep < RsStep < ModStep.

eq addSwc(C,rel N,H) = <AddSwcStep| todo, eta:T, cycle:N, -abs, hook:H, zet, swc:C >.
eq addCom(D,C.P.PP,M,rel N,H) = <AddComStep| todo, node:_, eta:T,
  cycle:N, -abs, hook:H, zet, dir:D, prop:C.P.PP, m:M >.
eq addMsg(K,M,S,I,P,rel N,H) = <AddMsgStep| todo, node:_, eta:T,
  cycle:N, abs:false, hook:H, zet:true, kind:K, m:M, size:S, route:I, peer:P >.
eq addIO(D,C.P.PP,IO,rel N,H) = <AddIomStep| todo, node:_, eta:T,
  cycle:N, -abs, hook:H, zet, dir:D, prop:C.P.PP, io:IO >.

eq rcCompleteEffect(addSwc(C), <A:Agent| swcs:CS > REST) = <A:Agent| swcs:C,CS> REST.
eq rcCompleteEffect(addCom(in,C.P.PP,M), <A:Agent|> <C:AppContainer|> REST)
  = <A:Agent| iMap[C.P.PP]=M, iChk[C.P.PP]=f> <C:AppContainer| comM[P.PP]=M> REST.
eq rcCompleteEffect(addCom(out,C.P.PP,M), <A:Agent|> <C:AppContainer|> REST)
  = <A:Agent| oMap[C.P.PP]=M, oChk[C.P.PP]=f> <C:AppContainer| comM[P.PP]=M> REST.
eq rcCompleteEffect(addIO(in,C.P.PP,IO), <A:Agent|> <C:AppContainer|> REST)
  = <A:Agent| iMap[C.P.PP]=IO, iChk[C.P.PP]=f> <C:AppContainer| comM[P.PP]=IO> REST.
eq rcCompleteEffect(addIO(out,C.P.PP,IO), <A:Agent|> <C:AppContainer|> REST)
  = <A:Agent| oMap[C.P.PP]=IO, oChk[C.P.PP]=f> <C:AppContainer| comM[P.PP]=IO> REST.
eq rcCompleteEffect(addMsg(local,m), <A:Agent|> REST) = <A:Agent| local+=M> REST.
eq rcCompleteEffect( addMsg(egress, m:M, size:N, route:I, peer:N2.I2), <A:Agent|> REST)
  = <A:Agent| egress[M]=I> setMsgQuota(I,M,N2.I2,N) REST.
eq rcCompleteEffect( addMsg(ingress, m:M, size:N, route:I), <A:Agent|> REST)
  = <A:Agent |ingress[M]=I> REST.
```

LISTING 4.6: The restructuring steps in Maude. The variants for absolute timing and the counterparts for removing are defined similarly but left out for brevity.

The sort *CycleSpec* is defined to make the timing of the modification steps more readable. Each of the constructor functions has a *CycleSpec* argument and sets both the *cycle* and *abs* accordingly. If $abs(N)$ is given, then *abs* is set to true and if $rel(N)$ is given, then *abs* is set to false. Only constructor functions for the relative variant are provided in listing 4.6. N is the value for *cycle*, which is consequently interpreted as absolute cycle number or relative to the most recent synchronization point. For all restructuring steps *zet* is set to true and *eta* is set to a constant value of $T = 1\,\text{ms}$, which represents the WCET of any kind of *RsStep* (the value of this constant is platform-specific, though). While restructuring steps could be performed in background, they should be performed in ZET mode to leverage the temporary quiescence guaranteed during reconfiguration hooks. Restructuring during full operation is only possible if not 'to many' reconfiguration steps need to be done in the same cycle turnover. This can be checked as described in section 4.2.

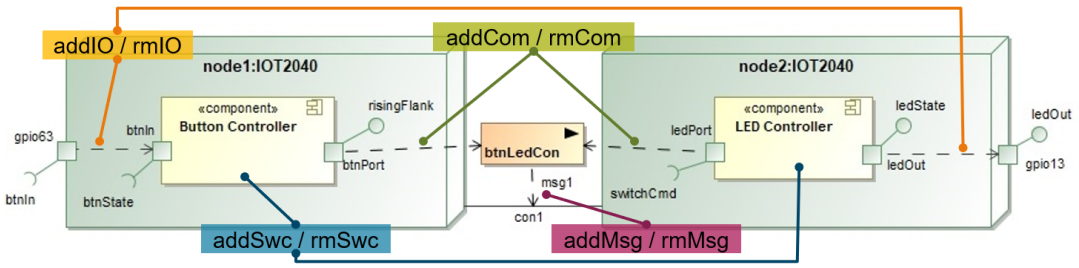

FIGURE 4.7: An overview of which parts of the system structure are modified by the defined restructuring steps at the example of the *onBtnSwitch* deployment.

Figure 4.7 shows an overview of the *RsSteps*. There are two restructuring steps to modify the SWC mapping: $addSwc(C, s, h)$ enables the task of software component C at the time specified by the *CycleSpec* c and the *RcHook* h; the inverse step $rmSwc(C, s, h)$ deactivates the task. As the add/remove pairs of restructuring have the exact same fields and the inverse *rcCompleteEffect*, only the definitions for adding are provided in listing 4.6. The corresponding *rcCompleteEffect* equations are applied by the generic [zetDone] rule for *ModSteps* with $zet = true$ as soon after *eta* expires as described before. The equation for *AddSwcSteps* adds C to the *Agent*'s *swcs* field, whereas the equation for *RmSwcSteps* removes it. Only activated tasks (of components in the *swcs* list) are triggered in the [setupCycle] rule of the *Agent* before each execution phase and compute values for their output properties from the values of their input properties. The component's container must have been started before via $start(C)$, so that the component is already initialized and ready to work. The input properties of C are wired beforehand using the corresponding restructuring steps, so that the task receives fresh values. To modify the communication specification two more restructuring steps are defined. A new message can be configured using $addMsg(k, m, size, dev_{route}, dev_{peer}, c, h)$. The *MsgKind* $k \in \{local, ingress, egress\}$ specifies whether the message is a local message (sender and receiver are deployed to the same node) or a network message – *ingress* is for incoming network messages and *egress* for outgoing network messages. The new message's identifier is m and the network bandwidth to allocate is defined by *size*. Additionally, two network interface identifiers dev_{route} and dev_{peer} are needed. For local messages, both devices are the same and refer to a local virtual interface, e.g. *br0*. For network messages, dev_{route} is the local network interface through which the message is sent (for *egress*) or received (for *ingress*) and dev_{peer} is the identifier of the remote network interface. This information is needed for configuring routing, traffic shaping and firewall – in this model represented by the *iMap*, *oMap* and the *local* fields of the *Agent*, the *comM* field of the *Containers* and the *qdiscs* field of network interfaces. The message is added by creating an entry in the corresponding field of the *Agent* (i.e., *ingress*, *egress*, *local*). Additionally, quota must be reserved for *egress* messages by sending a *setMsgQuota* command message to the network interface. A message can be removed from the communication specification via a *rmMsg* step using the same parameters; while in theory the message identifier would be sufficient, we avoid as much logic during time-critical reconfigurations as possible, so the parameters are included. The removal is implemented in the analog way by removing the corresponding message mappings. For *egress* messages the *Qdisc* quota is de-allocated via a *setMsgQuota* command message.

Using $addCom(d, p, m, c, h)$ and $addIO(d, p, \iota, c, h)$, properties of started components can be mapped to existing messages or I/Os. *Direction* $d \in \{in, out\}$ defines

whether the property is sent (*out*) or received (*in*). The qualified identifier $p = C.P.pp$ identifies the property pp of the component C's port P, while m identifies the message. The *Container*'s *comM* field is then modified accordingly by the coresponding *rcCompleteEffect* equations to configure whether and as which property messages and I/Os are available inside. Using *addCom* a local communication mapping is created in *comM* to realize one side of a matching (sender or receiver of the property). If $d = in$, then the *Agent*'s *iMap* and *iChk* fields are modified in addition so that incoming network messages for m are expected and accepted. If $k = out$, then the *Agent*'s *oMap* for outgoing network messages is modified in addition, so that the *Container* may send such messages. In this platform-independent model this mechanism is independent from whether the message is local or a network message. For example, the step $addCom(out, btn_1^{1.0}.btnPort.^{\triangleleft}risingFlank_{bool}^{50ms}, m_1, 3, beforeOutputs)$ adds a communication mapping, so the Boolean output property *risingFlank* of the button controller $btn_1^{1.0}$ is periodically extracted and sent via message m_1 starting in the output processing phase of the third cycle after the previous synchronization point. This is ensured because the *Agent*'s input and output processing rules include the existence of the specified mapping entries. A communication mapping is removed using *rmCom* with the same parameters and stops processing of a property. Finally *addIO* adds an I/O mapping in a similar way: property $p = C.P.pp$ can be mapped to a GPIO ι of the local node using $addIO(d, p, \iota, c, h)$ – again with *Direction* d, *CycleSpec* c and *RcHook* h. Using an I/O mapping an output property can be added for extraction and passing of the value from the software component C to the output ι, or conversely, for injecting the value from an input ι to a software component via the given input property. This is achieved by the mappings due to the definition of the [betweenIO], [in2swc] and [send2out] rules of the *Agent*, which trigger and process inputs and outputs accordingly. Again, I/O mappings can be removed using *rmIO* with the same parameters.

Handover Steps

Handover steps perform state transfer actions, which set up the internal states (see section 3.1.1) of new components using state information from running or removed components. This is often needed to maintain application consistency during reconfigurations, but it costs time. As we also want to maintain the reactivity of the embedded application we cannot block exhaustively to ensure consistency, though. This again leads to the problem that the states of the involved components and the transferred state snapshot might drift and become obsolete (i.e., the state of a running component changes concurrently to the transfer). Hence, the state transfer is time-cricital in two ways: 1) It must be performed in deterministic time so that the reconfiguration timing across nodes works without further coordination steps in time-critical reconfiguration phases. 2) It must be fast so that we can avoid blocking and state drift to maintain both reactivity and consistency – whenever possible, state transfer must be done in ZET mode. In addition to timing problems, we still have to isolate application-specific logic needed for state serialization and transformation using containers to avoid uncontrolled functional and non-functional behavior. All these aspects were considered during the definition of the component model so that consistent reconfigurations (including state transfer) can be achieved with minimal temporary degradation of the system's reactivity. Basic enablers for the handover concept are the *ModStep* concept, the different state kinds (stateless, ground state, implicit state), and declaration of the worst-case size of the state information (see

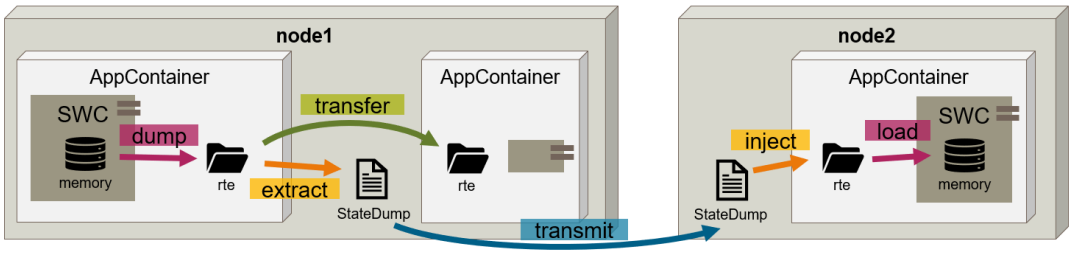

FIGURE 4.8: An overview of the defined handover steps. Basically, each step 'moves' state information between different state representations and locations.

section 3.1.1). The handover steps defined in the following (see listing 4.7 and figure 4.8) play a key role in enabling time-critical state transfers during operation of distributed embedded applications in the context of the other goals:

***dump*:** Let a component externalize its internal state.
***transfer*:** Directly copy state information from one component to another one.
***extract*:** Save the state of a component.
***transmit*:** Send state information to another node.
***inject*:** Pass state information to a component.
***load*:** Let a component load injected state information.

The steps *transfer*, *extract*, *transmit* and *inject* operate on persisted externalized state information objects of type *StateDump*. The structure of the state information is given by an application-specific subtype, but it must be *Serializable*, i.e., it must have a *size* field. In case of ground-state-components the state must be a *StateDump* object within the *Container*, which is accepted and actualized by the component in every cycle (see section 3.1.1). The worst-case size of the dumped state artifact is declared by the component in its state kind declaration. The ground state can be copied or sent to compatible ground-state-components directly with these four handover steps. Using *dump* and *load*, components with implicit state kind can be prompted to once produce/consume such a *StateDump*, so the state transfer can then be performed in the same structure-agnositic way.

The handover step $dump(C, d, eta, c, hook)$ can be used to let component C dump its internal state to a *StateDump* object d within its container. This operation must be implemented by components with implicit-state kind. With the inverse step $load(C, d, eta, c, hook)$ a *StateDump* d available inside of the container can be loaded to the internal state of component C. The WCET *eta* of the dump and load operations are declared by the component in its state kind declaration. The *CycleSpec* c and the *hook* specify the instant in which the steps are triggered. For *dump* and *load* the *zet* flag is always *false*, i.e., these two steps can only be performed in background. The dedicated *rcTriggerEffect* equation with the *isSwcStep* condition applies, which sends a *job* message to the corresponding *AppContainer* C instead of the *BgContainer*. Thus, *dump* and *load* steps are processed within the container of the corresponding component. As the container's *quota* is then used for these steps, the component's task must be deactivated using *rmSwc*, before. An *AppContainer*, which is not in the agent's *swcs* list, but in the set *bgm* of ongoing reconfiguration steps is still activated in the [setupCycle] rule by the *triggerBg* operation. The *AppContainer* gets CPU time during the subsequent cycles as of its quota until the *eta* of the reconfiguration step has run down to zero as of the *timeEffect* equation defined in listing 4.4. Then the [dmDone] rule for *dump* or the [ldDone] rule for *load* applies the effect of dumping

```
class HoStep | swc: Qid, id: Qid.          class StateDump < Serializable.
classes DumpStep LoadStep.                 classes ExtractStep InjectStep | ref: Qid.
class TransferStep | to: Qid, as: Qid.     class TransmitStep | to: Qid.
subclasses DumpStep LoadStep ExtractStep InjectStep TransferStep TransmitStep
  < HoStep < ModStep.

eq dump(C,D,E,rel N,H) = <DumpStep| todo, eta:E, cycle:N, -abs, hook:H, -zet, swc:C,id:D>.
eq load(C,D,E,rel N,H) = <LoadStep| todo, eta:E, cycle:N, -abs, hook:H, -zet, swc:C, id:D>.
eq extract(C,D,D2,Z,E,rel N,H) =
  <ExtractStep| todo, node:_, eta:E, cycle:N, -abs, hook:H, zet:Z, swc:C, id:D, ref:D2>.
eq inject(D2,C,D,Z,E,rel N,H) =
  <InjectStep| todo, node:_, eta:E, cycle:N, -abs, hook:H, zet:Z, swc:C, id:D, ref:D2>.
eq transfer(C,D,C2,D2,Z,E,rel N,H) =
  <TransferStep| todo, node:_, eta:E, cycle:N, -abs, hook:H, zet:Z, swc:C, id:D, to:C2, as:D2>.
eq transmit(C,D,NO,E,rel N,H) =
  <TransmitStep| todo, node:_, eta:E, cycle:N, -abs, hook:H, -zet, swc:C, id:D, to:NO>.

ceq if isSwcStep(<S|>) then rcTriggerEffect(<S:HoStep| swc:C, -zet>, <A:Agent| bgm> REST)
  = job(C,<S|>) <A:Agent| bgm[<S|>]=C > REST.
rl[dmDone]: <C:AppContainer| meta:V, memory:M, job:dump(ongoing,id:D,eta:0), rte:REST>
  => <C:AppContainer| over, job:_, rte:REST doDump(V,D,M)> jobDone(<DumpStep|>).
rl[ldDone]:<C:AppContainer| meta:V, job:load(ongoing,id:D,eta:0), rte:<D:StateDump|> REST>
  => <C:AppContainer| over, job:_, memory:doLoad(V,<D|>) > jobDone(<LoadStep|>).
eq rcCompleteEffect(<S:DumpStep|>, <A:Agent|> REST) = <A:Agent|> REST.
eq rcCompleteEffect(<S:LoadStep|>, <A:Agent|> REST) = <A:Agent|> REST.

eq rcCompleteEffect( extract(swc:C, id:D, ref:S), <C:AppContainer| rte:<D:StateDump|>> REST)
  = <C:AppContainer|> clone(S,<D:StateDump|>) REST.
eq rcCompleteEffect( inject(swc:C, id:D, ref:S), <S:StateDump|> <C:AppContainer| rte:R1 > REST)
  = <C:AppContainer| rte: writeDump( clone(D,<S:StateDump|>), R1) > REST.
eq rcCompleteEffect( transfer(swc:C, id:D, to:C', as:S),
  <C:AppContainer| rte:<D:StateDump|> R1> <C':AppContainer|rte:R2> REST)
  = <C:AppContainer|> <C':AppContainer| rte: writeDump( clone(S, <D:StateDump|>), R2)> REST.

eq rcTriggerEffect( transmit(swc:C, id:D, to:N), <A:Agent|oSync[N]=M, egress[M]=I, bgc:B> REST)
  = <A:Agent| bgm[<S|>]=B> tx(I,M,getState(C,D,R)) awaitTx(I,M,getState(C,D,R))) REST.
rl[transmitDone]: <N:NodeR| txDone(I,M,<D:StateDump|>) awaitTx(I,M,<D:StateDump|>)
  <A:Agent| egress[M]=I, bgm[<S:TransmitStep|id:D>]=B > REST>
  => <N:NodeR| rcCompleteEffect(<S:TransmitStep|>, <A:Agent| rm(bgm,S)> REST)>.
eq rcCompleteEffect(<S:TransmitStep| swc:null, id:D>, <D:StateDump|> REST) = REST.
eq rcCompleteEffect(<S:TransmitStep| swc:C>, REST) = REST [owise].
rl [acceptState]: rcv(I,M,<D:StateDump|>) <A:Agent| iSync[M]=N, ingress[M]=I >
  => <A:Agent|> <D:StateDump|>.
```

LISTING 4.7: The definition of handover steps in Maude.

or loading and sends a *jobDone* message to the agent. The agent then removes the steps from the *bgm* map as specified in the [modBgDoneMsg] rule (see listing 4.5) applying the corresponding *rcCompleteEffect* equations for *dump* and *load* – as the effect has already been applied before, these generic operations do not change anything more. Dumping is done by applying the *doDump* operation to the internal state of the component, i.e., the *memory* of the container, which creates *StateDump d* in the container's *rte*. The dump operation is component-specific, so each component with internal state has to define an equation for *doDump* which matches the unique versioned component type identifier set in the *meta* field. Loading is achieved similarly by applying the component-specific *doLoad* equation, which sets the valuations of the *memory* according to the contents of the given *StateDump d* taken from the container's *rte*. Of course, the structure of *d* must be understood by *C*'s *doLoad* operation, so *d* must have been provided by a compatible component. There is no specific handover step for transforming a *StateDump*. Instead, an update package component can be deployed temporarily to transform the state information if needed as proposed in [TK19b]. The temporal bounds of *dump* and *load* can be calculated as described beforehand based on the *quota* of the container. Per default the *quota* is configured according to the task's *wcet*, i.e., the *StateDump d* is ready/processed after $n = \lceil eta/wcet \rceil$ cycles.

A *StateDump d* can be copied from the *rte* of container C_1 to the *rte* of a co-located

container C_2 as d_t using $transfer(C_1, d, C_2, d_t, zet, eta, c, h)$. As for most modification steps, if *zet* is true, then the step is performed in ZET mode directly within the reconfiguration hook h in the cycle determined by c. Otherwise, the step is only triggered at that time and performed in the *BgContainer*. This is specified by the generic *ModStep* triggering equations. In ZET mode the components are quiescent due to execution within the reconfiguration hook. In background mode it must be ensured during design of the reconfiguration plan that C_1 does not modify d and C_2 does not use d_t before the step's temporal bound expires. For example, this is the case, if C_1 is blocked or has implicit-state kind and C_2 is only activated after the tranferral. To estimate *eta*, the worst-case state size s is needed (as declared by the providing component) in addition to the available memory I/O quota *blkio*[1]: The estimated execution time is $s/blkio$. If performed in background, the step completes within $n = \lceil s/blkio_{bg} \rceil$ cycles, with the background workers memory I/O quota $blkio_{bg}$. Finally, this bound is projected to the background worker's CPU quota Q_{bg} by setting $eta = n \times Q_{bg}$. However, independently of how the step is triggered, the corresponding *rcCompleteEffect* for *transfer* is applied to create the copy of the *StateDump* as soon as *eta* expires either due to the [zetDone] rule or the [modBgDoneMsg]. To further avoid blocking and to enable more complex state transfers a "split" of the transfer step is defined: The handover step $extract(C, d, d_C, zet, eta, c, h)$ copies the *StateDump* d from the *rte* of container C to the outside inaccessible to components and renames it to d_C to avoid potential name conflicts. The opposite step $inject(d_C, C, d, zet, eta, c, h)$ copies *StateDump* d_C from the node-local outside to the *rte* of container C and renames it to d. For both steps the same rules and considerations for triggering, completion and runtime bounds apply.

Finally, the *transfer* step transmits a *StateDump* object to another node. This step is always executed in background (i.e., its *zet* is always *false*). Though this step is performed in the *BgContainer*, its triggering, execution and completion is modeled differently to achieve compatibility with the network model. There are two variants of the step, depending on the *swc* field: If it is set, then the *StateDump* is taken from the corresponding container's *rte*. Otherwise, it is taken from the node-local storage (e.g. after extraction from a container). This is done by the $getState(C, d, R)$ operation used in the *rcTriggerEffect* equation for *transmit* (R contains the container C and the *StateDump* d – either within or outside of C). Instead of sending a *job* message to the *BgContainer*, the *Agent* directly triggers the asynchronous transmission via a $tx(i, m, <d{:}StateDump \mid >)$ message. The network message m is taken as configured in *oSync* for inter-node communication with the node specified in the *to* field of the *TransmitStep*. The network interface i is configured in *egress* for routing traffic related to network message m. In addition to the *tx* message, a corrsponding *awaitTx* message is created to mark a continuation of the procedure after successful transmission. After the network model has transmitted the *StateDump* during the next cycles according to the *quota* configured for the *Qdisc* of m the network model creates a transmission acknowledge message *txDone* on the sending node as of the [rcv] rule, which is not discarded due to the *awaitTx* marker message. This is used as trigger for the [transmitDone] rule, which completes the step by removing it from the *bgm* map of ongoing reconfiguration steps. Additionally, the *StateDump* d is removed, if it was taken from the node-local storage. Otherwise, no further *rcCompleteEffect* is defined. On the receiving node, the *StateDump* d arrives at the same time within a $rcv(i', m, <d \mid >)$ message. The [acceptState] rule accepts and stores the d in the

[1]Memory I/O is not modeled in the scope of this thesis, but a model similar to the CPU bandwidth control is assumed. Therefore, *eta* and the *quota* are used to abstract from memory scheduling and its effect on resource utilization.

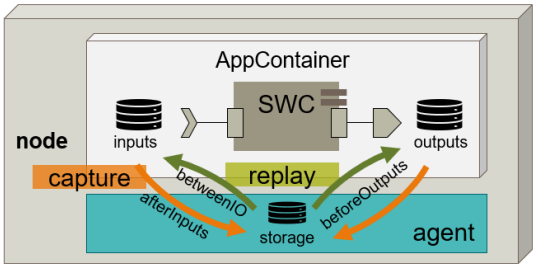

FIGURE 4.9: An overview of the defined interception steps. All required and provided properties can be captured and replayed each cycle during specific hooks.

node-local storage inaccessible to containers if an incoming network message m at network interface i' is configured in *ingress* and *iSync* directs m to the agent. To calculate the temporal bounds of the *transmit* step the worst-case size s of the *StateDump* is used as declared by the providing component. Additionally, the quota M_{adm} per period and the protocol overhead O_{adm} are needed (see section 3.1.4). The network connection transmits $s + O_{adm}$ bits with M_{adm} bits per period (enforced by the *quota* for m), so the transmission bound is $n = \lceil (s + O_{adm}) / M_{adm} \rceil$ cycles. Thus, the *eta* field of the *TransferStep* must be set to $n \times Q_{bg}$ in order to project n to the background worker's assigned execution time Q_{bg}. The *eta* field is not used for simulation of the *TransferStep*'s temporal behavior because the network model uses the *quota* and *size*. However, the *eta* field must still reflect the temporal bounds of this kind of reconfiguration step, too, because it is used for checking reconfigurations (see section 4.2)[2].

Interception Steps

Interception steps manipulate property valuations of running containers, i.e., inputs and outputs of components. This is needed to fix broken data processing and transmission chains in case of blocked components. In this platform-independent model it is irrelevant whether a property is mapped to a message or to an I/O. In the platform-specific realization, properties may need to be manipulated differently depending on their mapping. Following interception steps are defined (see listing 4.8 and figure 4.9):

***capture*:** Store the current valuation of the specified property.
***replay*:** Manipulate the valuation of the selected property.

The two interception steps *capture* and *replay* have two additional fields inherited from *IntStep* (besides the other *ModStep* fields): p refers to a qualified property and *id* holds a unique capture reference. The steps inherit the basic behavior for triggering, execution, and completion from *ModSteps*, but they can only be performed in ZET mode. Hence, the constructor functions for *capture* and *replay* set $zet = true$. The effects are specified by two *rcCompleteEffect* equations per step, which match depending on whether the corresponding property is an input or an output. The interception step $capture(p, ref, c, h)$ stores the current valuation of the qualified property $p = C.P.pp$ in the *storage* of the node-local agent. Reference *ref* is used to retrieve the value later. *RcHook* h and *CycleSpec* c determine the timing of the step. When executed, the current valuation is taken from the *inputs* or *outputs*

[2]Even though this calculation would be done by engineering tools, a more comprehensible bounds-indication should cover all the relevant resources (e.g. CPU, network, and memory). For the scope of this thesis we project all resources to *eta*, though.

```
class IntStep | p:QQid, id:Qid.
classes CaptureStep ReplayStep < IntStep < ModStep .
eq capture(C.P.PP,I,rel N,H)
  = <CaptureStep| todo, node:_, eta:T, cycle:N, -abs, hook:H, zet, p:C.P.PP, id:I>.
eq replay(C.P.PP,I,rel N,H)
  = <ReplayStep| todo, node:_, eta:T, cycle:N, -abs, hook:H, zet, p:C.P.PP, id:I>.
eq rcCompleteEffect( capture(C.P.PP,I), <A:Agent|> <C:AppContainer| inputs[TY P.PP]=V > REST)
  = <A:Agent| storage[TY I]=V > <C:AppContainer|> REST.
eq rcCompleteEffect( capture(C.P.PP,I), <A:Agent|> <C:AppContainer| outputs[TY P.PP]=V > REST)
  = <A:Agent| storage[TY I]=V > <C:AppContainer|> REST.
eq rcCompleteEffect( replay(C.P.PP,I),
    <A:Agent| storage[TY I]=V > <C:AppContainer| inputs[TY P.PP] > REST)
  = <A:Agent| iChk[C.P.PP]=y > <C:AppContainer| inputs[TY P.PP]=V > REST.
eq rcCompleteEffect( replay(C.P.PP,I),
    <A:Agent| storage[TY I]=V > <C:AppContainer| outputs[TY P.PP] > REST)
  = <A:Agent|> <C:AppContainer| outputs[TY P.PP]=V > REST.
```

LISTING 4.8: The definition of interception steps in Maude.

map of container C. The inverse operation $replay(p, ref, c, h)$ sets or overwrites the current valuation of the property p to the stored value referenced by *ref*. For example, the button controller's rising flank output can be captured and replayed with one cycle offset using $capture(^{\triangleleft}btn\text{-}v1.btnPort.risingFlank, flank, 0, beforeOutputs)$ followed by $replay(^{\triangleleft}btn\text{-}v1.btnPort.risingFlank, flank, 1, beforeOutputs)$. Output properties should be captured and replayed in the hook *beforeOutputs*. This ensures that the captured output is the one resulting from the previous task execution in case of a *capture* step. In case of a *replay* step, this ensures that the replayed output is extracted during the subsequent output processing phase directly following the hook *beforeOutputs*. Input properties should be replayed in the hook *betweenIO* and captured *afterInputs*. Replaying in the hook *betweenIO* ensures that the replayed input is not missed during input processing (because it is marked as received/provided in *iChk*) and that the input is available within the container in the subsequent execution phase. Capturing in the hook *afterInputs* ensures that the most recent input values are available within the container. However, please note that these hooks are required in this platform-independent model, only. Depending on the platform-specific implementation of capturing and replaying properties it might be necessary to place interception steps in different hooks. In this case, the hooks must be given in the platform-independent form to check the reconfiguration consistency as described in section 4.2. Finally, we propose that *replay* steps should only be used to fix broken chains, i.e., to provide values which are expected to miss (e.g. because of blocking). For other use cases such as testing and fault injection we suggest to define additional steps in future (e.g. *drop*, *override*, ...), which enable detection of erroneously missing inputs and outputs, for instance.

4.2 Reconfiguration Consistency

After specifying the individual reconfiguration step kinds the question arises: How do we define and check the consistency and feasibility of reconfiguration plans? During reconfiguration execution, the model transitions from the old/current to the new deployment version in a transactional manner instead of an instant switch-over. Thus, the runtime states on the different nodes do not continuously yield a valid deployment in terms of section 3.3. Moreover, the consistency criteria must be relaxed sometimes to enable certain reconfigurations with state transfer – we refer to this as temporary quality degradation (cf. [TK19b]). The reconfiguration consistency checks described in the following must use a different, more operational definition

of correctness and feasibility, therefore. For this purpose, a valid (as of section 3.3) design-time model (distributed embedded application, topology, deployment) and a corresponding instantiation of the runtime model are needed with one (potentially empty) reconfiguration plan for each node. Using the approaches described in this section an engineer can then systematically analyze whether the composed reconfiguration plan is valid. However, this analysis does not always result in a yes/no decision: Especially for the design of reconfigurations which include state transfer a structured quantitative analysis of the resulting quality degradation is needed. Given this information the engineer can compare alternative reconfiguration plans and decide which one best fulfills the potentially conflicting goals of ensuring consistency, feasibility and reactivity during reconfigurations.

4.2.1 Correctness and Feasibility

The first task is to ensure that the modeled reconfiguration steps are **correct**. Regarding the non-functional interface, the correctness of the *eta* field of modification steps is critical. For restructuring steps (e.g. *addSwc*) and interception steps (e.g. *capture*) these are platform- and MCU-specific constants that need to be identified once for a concrete board using repeated measurements with different benchmark deployments. The durations of handover steps depend on the step kind and field values. This was described in section 4.1 and is only briefly summarized here. The *eta* field for *dump* and *load* must be at least equal to the declared values in the state interface of the corresponding component. The steps *dump* and *load* are only supported by components with implicit-state kind. The *eta* of the 'copy' operations *extract*, *transfer* and *inject* depend on the worst-case size s (taken from the component's state interface) and the available I/O quota – if $zet = true$ then the agent's quota is used for the step, or otherwise the quota of the background worker. Both must be overestimated by platform-specific tables, which map state sizes and quota to WCETs. Additional platform-specific throughput tables are needed to overestimate the *eta* of *transmit* steps. This depends on the state size, the configured network bandwidth M_{adm} and also, on the CPU quota Q_{bg} of the background worker. Examples for such measurements can be found in Section 6.1.3 and in [TK19b]. The *eta* fields of the lifecycle steps are not critical: They are aligned using coordination steps, so the *eta* field is only used for simulation purposes and to estimate the duration of the reconfiguration. Coordination steps are assumed to take no time (ZET assumption).

For the remaining analysis we assume that the *eta* information is set in this way and, thus, correctly reflects the execution time of the steps. We can then achieve a well-defined temporal behavior of multi-node reconfigurations with following **basic rules for reconfiguration plans**: After each block of lifecycle steps there must be a block of coordination steps on all nodes which shall synchronously run a block of modification steps afterwards. The coordination steps must be defined according to the synchronization protocol, if multiple nodes are involved. Otherwise, an absolute or relative single-node coordination step must be specified before a block of modification steps. Absolute offsets (i.e., *waitForCycle* and `<ModStep| abs: true>`) are only allowed at the beginning of the reconfiguration plan, i.e., before lifecycle steps or synchronization protocol steps are used. As lifecycle steps are performed one after another in the background container and due to the subsequent coordination, their durations are irrelevant for the consistency of the reconfiguration. To handle their non-deterministic timing, we group synchronous blocks of lifecycle steps and of modification steps and sort the blocks by their preceding synchronization point. Blocks of reconfiguration steps happen synchronously on the involved nodes if they

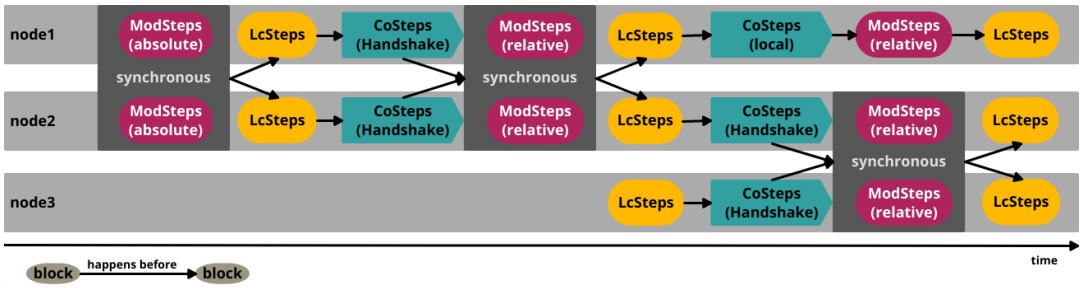

FIGURE 4.10: An illustration of synchronous and parallel blocks of reconfiguration steps. Blocks are synchronous if they have absolute cycle timing or if they are aligned using the handshake. Only synchronous blocks of modification steps can consistently create reconfiguration effects, which impact other nodes.

are placed directly after a handshake (synchronization protocol) or start at the same time due to their absolute cycle offsets. All other blocks are timed relative to the synchronous blocks, but without any further guarantee of timing, i.e., the blocks are partially ordered. Figure 4.10 shows an overview of the partial ordering of temporally unaligned blocks relative to the synchronous blocks.

The **logical consistency** of the reconfiguration plans now basically depends on whether the reconfiguration steps fit to the runtime situation at the time of their planned execution. Note that the different aspects described in the following are already incorporated in the model in section 4.1, but not all consistency requirements are explicitly tethered with the reconfiguration steps (e.g. schedulability when adding a component). To ensure the consistency of the distributed embedded application during temporally independent blocks of reconfiguration steps, such blocks must not create an externally visible reconfiguration effect. Therefore, only backwards-compatible modifications are allowed in such blocks, i.e., changes w.r.t. network-mapped properties are inhibited except for adding receivers of already existing messages. For example, no coordination across nodes is needed, when a running software component can be replaced with a completely compatible new version without downtime. But if the new component version uses different types for network-mapped properties (i.e., properties required or provided by components running on other nodes), then such a reconfiguration cannot be done in temporally independent blocks of modification steps – most likely it would lead to incompatible communication partners. To check whether the reconfiguration steps fit to the runtime situation at the corresponding time, we have to lock-step walk through the reconfiguration plans and ensure that the current runtime model fulfills the requirements of the specific reconfiguration steps. For **lifecycle steps** the following basic consistency requirements (i.e., preconditions) apply (see section 4.1.2):

- $download(meta, c)$: An image for *meta* exists in the component repository and no component instance c exists on the node.
- $start(c, quota)$: No container for c is running, yet. The *quota* becomes relevant, when the component is activated (see *addSwc*).
- $stop(c)$: The container for c is running, and there is no active task or ongoing reconfiguration of c.
- $destroy(c)$: The container for c exists and is stopped.

For **modification steps** some general consistency requirements must be met, first. If *zet* is false, then the job is sent either to the background container or to the corresponding app container (in case of *dump* and *load*). There must not be an active task

or another ongoing reconfiguration step in the corresponding container between the time of triggering a modification step and the time it is completed. This restriction ensures that the temporal bounds of the reconfiguration steps can be calculated from the container's *quota* and the step's *eta*, only. As described before, such a modification step then completes and provides the corresponding reconfiguration effect within n cycles, $n = \lceil eta/quota \rceil$. While this leads to bounded time, it does not utilize the temporary quiescence during reconfiguration hooks. Thus, only handover steps may be performed in background (with additional constraints as described below). For restructuring steps and interception steps the reconfiguration plan description language ensures this by setting $zet = true$. These steps are then performed within the administration phase, so we have to ensure that not too many steps are planned for each cycle. Subsequent modification steps within the reconfiguration plan of one node are planned for the same cycle, if their *cycle* attribute is equal. This is independent from *abs* and *hook*. For each sequence of modification steps planned for the same cycle, the subset of modification steps with $zet = true$ are contemporary – they are executed during the same administration phase on the same node. For each such set RC_{zet} of contemporary reconfiguration steps, the sum of the steps' *eta* must be less than or equal to the allocated time for the adminitration phase $WCET_{adm}$:

$$\sum_{step \in RC_{zet}} eta(step) \leq WCET_{adm}$$

Provided that these general requirements are met, following specific consistency requirements (i.e., preconditions) apply to the modification steps. For better readability we only include the parameters important for the reconfiguration effect. Note that these preconditions are incorporated in Maude rules and equations in section 4.1 and can be checked by simulation as described later in this section.

- $addSwc(c)$: Component instance c is deactivated but its container is running and there is enough free CPU space for its *quota* on the corresponding node.
- $rmSwc(c)$: Component instance c is activated.
- $addMsg(k, m, size, dev_{route}, dev_{peer})$: No message m exists. The two devices dev_{route} and dev_{peer} exist. In case of a local message ($k = local$) both must be equal and refer to a network interface of the local node (in the agent's *ifs* set). Otherwise ($k \in \{ingress, egress\}$) only dev_{route} is a local network interface, while dev_{peer} must be a network interface directly connected to dev_{route}. There must be enough bi-directionally guaranteed bandwidth left on the corresponding network connection.
- $rmMsg(k, m, size, dev_{route}, dev_{peer})$: Message m must exist on the local node and must not be used by a communication mapping.
- $addCom(d, p, m)$: The property p and message m exist, m's *size* is sufficient for the additional property, and its *kind* matches the direction d and the interface of p (required/provided). No such mapping exists, yet.
- $rmCom(d, p, m)$: Such a communication mapping exists.
- $addIO(d, p, \iota)$: No I/O mapping exists for ι and the software ports match.
- $rmIO(d, p, \iota)$: Such an I/O mapping exists.
- $dump(c, d)$: Component instance c is deactivated, but its container is running. No other reconfiguration step is performed by c as of the *bgm* of the agent.
- $load(c, d)$: Like *dump*, but additionally, the *StateDump* artifact d must exist within the container of c.

- *transfer*(c_1, d_1, c_2, d_2): Containers for component instances c_1 and c_2 must exist (but can be stopped or running). The *StateDump* artifact d must exist within the container of c_1.
- *extract*(c, d, d_c): The container for c must exist with *StateDump* artifact d inside.
- *inject*(d_c, c, d): The container for c must exist and *StateDump* d must exist in the node-local *storage*.
- *transmit*(c, d, n): If $d \neq null$ then a *StateDump* d must exist within the container c. Otherwise d must exist in the node-local *storage*. There must be a valid *oSync* message for inter-agent communication with node n on the sending node and an *iSync* message on the receiving node, as well as on intermediary nodes.
- *capture*$(c.p.pp, r, h)$: The container for component instance c with port p and property pp must be running. If pp is a provided property, then the reconfiguration hook h should be set to *beforeOutputs* or *betweenIO* (platform-specific). In this case c must have been active in the previous cycle. If it is a required property, then h should be *afterInputs* or *betwenIO* (platform-specific).
- *replay*$(c.p.pp, r, h)$: Like for *capture*, c and pp must exist. If pp is a provided property, then h should be *beforeOutputs* or *betweenIO*. If it is a required property, then h should be *betweenIO* or *afterInputs*. After replaying a required property, c should be activated to use the value. Provided properties should only be replayed when c had been deactivated in the previous cycle. A captured value for r with the appropriate type for pp must exist within the node-local *storage*.

To ensure that the **state artifacts** required by handover steps will exist at the scheduled time, we have to consider the timing within each synchronous block of modification steps. For implicit-state components, *dump* must be completed (as of its temporal bound) before the *StateDump* can be moved using *extract*, *transfer* or *transmit*. This basically applies to all chains of handover steps which gradually 'move' a state. For instance, a *transmit* step must be completed before the transmitted *StateDump* can be *injected* to the component on the receiving node. Additionally, we have to consider the **dataflow consistency** of the evolving distributed embedded application. During the reconfiguration, each activated software component must continuously receive new (w.r.t. *maxAge*) input values for their required properties (either from a sensor, from another component, or via a *replay*). All provided properties mapped to an actuator or message must also hold their *refreshmentPeriod* (either by the activated task or via a *replay*).

All these consistency aspects can simply be **validated by simulation** using Maude: The pre-conditions of the reconfiguration steps are specified in the corresponding rules, so the rules only match when the pre-conditions are fulfilled. Additionally, the model is constructed using *mte* functions so that time cannot pass if an applicable rule must be applied (as of its trigger condition specified by *mustRc*). This also applies to CPU and network scheduling as well as to system management tasks beyond reconfiguration steps during the administration phase. Thus, we can check the consistency and feasibility of a reconfiguration plan and of the resulting new deployment with a simulated run of the reconfiguration. If the simulation hangs at any time because progression of time is inhibited, then the reconfiguration plan or the new system are inconsistent or not feasible. The engineer can then inspect the trace of the simulation and check the agent states and find out what prevents further progression of time. If the agent hangs in a reconfiguration hook, then because the step at the head of the node-local reconfiguration plan must be applied, but its precondition is not fulfilled. If the agent hangs in input or output processing, then an input or output is missing or duplicated, which can be seen in the *iChk* and *oChk* maps. In

this case the engineer can check the trace or the resulting runtime model configuration to break down this issue. For instance, the reason for a missing output probably is that the corresponding task did not complete its cycle (e.g. because it was not activated, because the container was started with too little *quota*, because the state was inconsistent, ...). An input can be missing if the message could not be transmitted in time or because no corresponding mapping was created. All these issues can be revealed by a hanging simulated run, either because the reconfiguration would not be completed or the resulting system would not work. Thus, a reconfiguration plan and the resulting deployment are consistent and feasible if the simulation reaches the end of the reconfiguration plan and then successfully completes one end-to-end simulation (i.e., covering the resulting distributed control system's *wcrt*). Please note that this consistency check only covers the functional and non-functional aspects of the distributed control system itself; the technical process requirements are beyond the scope of this thesis (e.g. reachability of hazards).

4.2.2 Quality Degradation

Two kinds of temporary quality degradation of the distributed control system may be needed to achieve consistency of the dataflow while performing a time-consuming state transfer:

- Delayed reaction: The worst-case reaction time from measured inputs to calculated outputs might temporarily exceed the *wcrt* limit specified by the distributed embedded application.
- Temporary blindness: Measurable events in the technical process could be missed due to temporary blocking.

Quality degradation is possible even when the reconfiguration plan is consistent and feasible as defined before. It could be completely avoided by inhibiting reconfiguration plans, which would (temporarily) break data processing and transmission chains (i.e., which remove I/O mappings, modify a path from sensors to actuators, or patch a path with *capture* and *replay*). However, this would make many distributed control systems non-reconfigurable because state transfer can take time. Therefore, we describe how to analyze the reactivity during reconfigurations using the **evolving dataflow graph** (EDFG). An engineer can then decide whether the resulting temporarily degraded reactivity is acceptable. We construct the dataflow graph by sweeping through system execution during a synchronous block of reconfiguration steps time at specific instants, i.e., in reconfiguration hooks (when reconfiguration steps are triggered or completed) and between them (when the model is well-defined). Thus, there are six instants in each cycle in which we add nodes and edges to the dataflow graph: Before output processing, during output processing, between output and input processing, during input processing, after input processing and during execution. The graph construction procedure described below starts before output processing at the synchronization point in relative cycle $n = 0$ and sweeps through the six instants of all cycles until the reconfiguration block is completed. During reconfiguration hooks we add nodes and edges for the dataflow caused by the applicable handover steps and interception steps. During output processing, input processing and execution, we add nodes and edges for the dataflow caused by the system execution itself according to the current runtime model at the given instants. All nodes are labelled with the cycle to distinguish between cyclic data points at different times. Additionally, edges are weighted with time, so the path lengths from sensors to actuators represent the current reaction times.

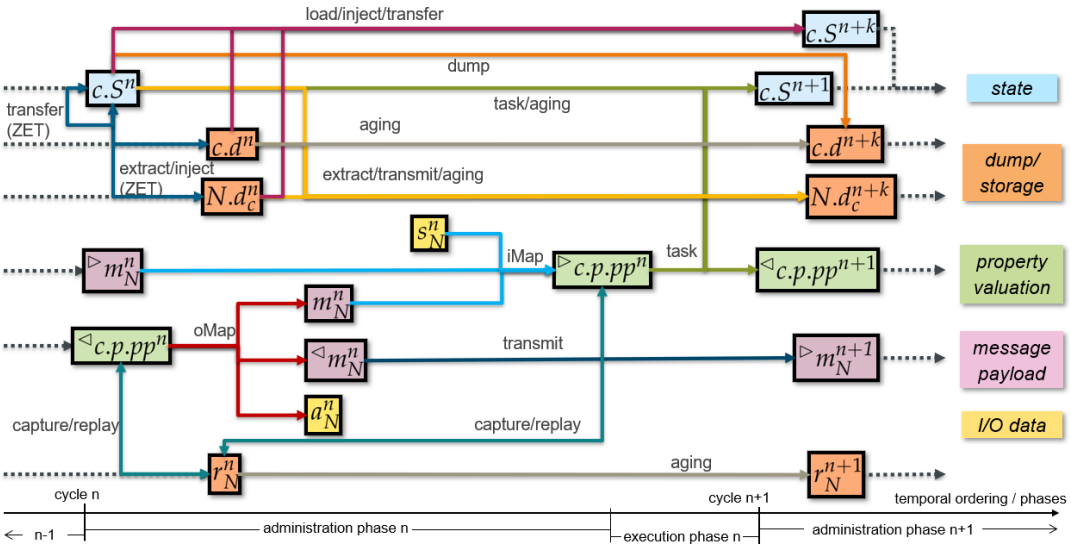

FIGURE 4.11: Schematic visualization of the nodes (cf. table 4.1) and edges potentially added to the evolving dataflow graph during construction.

node	description
$c.S^n$	the combined state of component c at the beginning of cycle n (including ground state and internal state)
$c.d^n$	state dump d of/for implicit-state component c created/available in cycle n
$N.d_c^n$	extracted state dump d of component c available on node N in cycle n
${}^{\triangleright}m_N^n$	network message m received on node N for processing in cycle n
${}^{\triangleleft}m_N^n$	network message m sent from node N in cycle n
m_N^n	local message m on node N in sent and received in cycle turnover n
${}^{\triangleright}c.p.pp^n$	input for required property pp of component c, port p in cycle n
${}^{\triangleleft}c.p.pp^n$	output for provided property pp of component c, port p, available in cycle n (i.e., calculated in cycle $n-1$)
r_N^n	capture entry created/available on node N in cycle n
s_N^n	input of sensor s on node N in cycle n
a_N^n	output for actuator a on node N in cycle n

TABLE 4.1: Short description of the nodes within the evolving dataflow graph.

Figure 4.11 shows a schematic overview of the nodes and edges in the evolving dataflow graph. The nodes and edges related to the normal application dataflow are basically equal to the non-evolvable dataflow graph described in section 3.3. However, the nodes are labelled with the cycle and the nodes and edges are only inserted conditionally, i.e., depending on the state of the runtime model at the corresponding instants. We are **constructing the graph** for a synchronous block of modification steps during system execution. If this block is at the absolute cycle $n = 0$ then we add nodes for the initial inputs, outputs, and states (in fact, a start-up reconfiguration must be employed as described in section 4.3.4). Otherwise, the data from the previous cycles is available and we add them to the **initial graph** for the relative cycle $n = 0$: For each stateful component c we add a state node $c.S^0$ for its initial state resulting from the previous execution phase. For each input message $m \mapsto I \in ingress$ on MCU N we add a reception node ${}^{\triangleright}m_N^0$ representing the data received during the previous execution phase. For each provided property ${}^{\triangleleft}c.p.pp$ of a component $c \in swcs$ we add an output node ${}^{\triangleleft}c.p.pp^0$ representing the task outputs from the previous execution phase. After initializing the graph in this way, we add

the application dataflow according to the actual model states, as they are subject to change by modification steps. In each **output processing** phase, we add nodes and edges for the active output mappings in *oMap*. We add a new bridge node m_N^n for each provided property ${}^{\triangleleft}c.p.pp$, which is mapped to a local message $m \in local$ on MCU N in cycle n. We represent the dataflow of by an edge from the output node ${}^{\triangleleft}c.p.pp^n$ to the bridge node m_N^n with weight 0. Similarly, we add a new sending node ${}^{\triangleleft}m_N^n$ for each provided property ${}^{\triangleleft}c.p.pp$, which is mapped to an outgoing network message $m \mapsto I \in egress$ on MCU N. The dataflow is represented by an edge from output node ${}^{\triangleleft}c.p.pp^n$ to sending node ${}^{\triangleleft}m_N^n$ with weight 0. For each provided property ${}^{\triangleleft}c.p.pp$ mapped to an actuator $a \in actuators$ in cycle n we add an actuator node a_N^n and an edge from output node ${}^{\triangleleft}c.p.pp^n$ to a_N^n. The weight of such an edge is set to the *refreshmentPeriod* declared by the I/O port. In the **input processing** phases, we add nodes and edges for the active input mappings in *iMap*. For each required property ${}^{\triangleright}c.p.pp$ mapped to a sensor $s \in sensors$ in cycle n we add a sensor node s_N^n for the sensor input, an input node ${}^{\triangleright}c.p.pp^n$ for the component input and a dataflow edge from sensor node s_N^n to input node ${}^{\triangleright}c.p.pp^n$. The weight of such an edge is set to the *maxAge* declared by the I/O port. We add a new input node ${}^{\triangleright}c.p.pp^n$ for each required property ${}^{\triangleright}c.p.pp$ mapped to a local message $m \in local$ on MCU N in cycle n. The dataflow of the local message is represented by an edge from bridge node m_N^n to input node ${}^{\triangleright}c.p.pp^n$ with weight 0. Similarly, we add a new input node ${}^{\triangleright}c.p.pp^n$ for each required property ${}^{\triangleright}c.p.pp$ mapped to a network message $m \mapsto I \in ingress$ on MCU N. The dataflow is modeled by an edge from reception node ${}^{\triangleright}m_N^n$ to input node ${}^{\triangleright}c.p.pp^n$ with weight 0. In the **execution phase** of cycle n, we represent the dataflow for the calculation in each active task by adding following elements. We add output nodes ${}^{\triangleleft}c.p.pp^{n+1}$ to represent the output data of each provided property of each component $c \in swcs$. Then for each pair of required and provided properties of an active component $c \in swcs$ we add an edge from input node ${}^{\triangleright}c.p_a.pp_i{}^n$ to output node ${}^{\triangleleft}c.p_b.pp_j{}^{n+1}$ to model the dataflow caused by calculating the outputs from the inputs. If component c has declared a ground state or an implicit state, then we add a state node $c.S^{n+1}$ and dataflow edges from the 'old' state node $c.S^n$ to the 'new' state node $c.S^{n+1}$ as well as edges from input nodes ${}^{\triangleright}c.p_a.pp_i{}^n$ to the 'new' state node $c.S^{n+1}$ and from the 'old' state node $c.S^n$ to each output node ${}^{\triangleleft}c.p_b.pp_j{}^{n+1}$. All these edges have weight *period* because the calculation takes one period of locical execution time. If a stateful component is deactivated, then we only add an "aging" edge from the 'old' state node $c.S^n$ to the 'new' state node $c.S^{n+1}$ with weight *period*. Additionally, we add nodes and edges for the network communication during the execution phase. For each activated hop from MCU N_1 to N_2 of each network message m we add a reception node ${}^{\triangleright}m_{N_2}^{n+1}$ and a dataflow edge from sending node ${}^{\triangleleft}m_{N_1}^n$ to reception node ${}^{\triangleright}m_{N_2}^{n+1}$. Such a hop is activated during an execution phase if the agent on N_i has m in its *egress* map, the agent on N_2 has m in its *ingress* map, and both maps point to directly connected network interfaces (they are consecutive in m's *path*). This is the case if m has already existed or has been added with *addMsg* (not been removed) on both nodes before that execution phase. As the transmission takes one cycle of logical execution time, the weight of this edge is also set to *period*.

Besides the normal application data flow, there is additional dataflow caused by handover steps and interception steps. For each $capture(c.p.pp, r)$ step applied in cycle n we add a capture node r^n for the capture entry. If pp is a provided property, then we add an edge from output node ${}^{\triangleleft}c.p.pp^n$ to capture node r^n. If it is a required property, then we add an edge from input node ${}^{\triangleright}c.p.pp^n$ to capture node r^n. The weight of such an edge is 0. For this analysis, *capture* steps for provided properties must be placed before output processing and *capture* steps for required properties

must be placed after input processing, even if platform-specific realizations require different hooks as described before. For a $replay(c.p.pp, r)$ step we add an edge from capture node r^n with weight 0. If it is a provided property, then the dataflow edge goes to output node ${}^{\triangleleft}c.p.pp^n$. As the component must be deactivated in this case it is ensured that this node is only reachable via the replay edge. If a required property is replayed, then the edge goes to input node ${}^{\triangleright}c.p.pp^n$. Again, this node must be reachable via the replay edge, only, which in this case is ensured if there is no active mapping for that input during input processing in cycle n. Next, we consider the handover steps. For each $transfer(c_1, d_1, c_2, d_2)$ step triggered in cycle n we add an edge from the 'current' dump node $c_1.d_1^n$ of implicit-state component c_1 to the 'future' dump node $c_2.d_2^{n+k}$ of implicit-state component c_2 (representing the *StateDump* d_2 of component c_2 in cycle $n + k$). The parameter k depends on how long the step takes. If *zet* is *true*, then we set $k = 0$, because the step is completed within the reconfiguration hook in cycle n. Otherwise we set $k = \lceil eta/quota \rceil$, which is the bound as described before. This models the property that the transferred state is available for c_2 after the step completed in background k cycles later. Therefore, the weight of such an edge is $k \cdot period$. This applies to the following steps, too, whenever k is used. If c_1 is a component with ground state kind, then the *transfer* edge starts from state node $c_1.S^n$ instead of dump node $c_1.d_1$, because in this case the *StateDump* belongs to the ground state. For the same reason, if c_2 is a component with ground state kind, then the *transfer* edge goes to the 'future' state node $c_2.S^{n+k}$ instead of dump node $c_2.d_2^{n+k}$. For each $extract(c, d, d_c)$ step on node N in cycle n we add a new dump node $N.d_c^{n+k}$ and an edge from the 'current' dump node $c.d^n$ to the 'future' dump node $N.d_c^{n+k}$. Again, if c has ground state kind, then the edge starts from state node $c.S^n$ instead of dump node $c.d^n$. For each $inject(d_c, c, d)$ step on node N in cycle n we create a 'future' dump node $c.d^{n+k}$ for the injected *StateDump* in cycle $n + k$ and add an edge from the 'current' dump node $N.d_c^n$ to the 'future' dump node $c.d^{n+k}$. If component c is a ground state component, then the edge goes to the component's 'future' state node $c.S^{n+k}$ instead of a dump node $c.d^{n+k}$. For each $transmit(c, d, N_2)$ step on node N_1 in cycle n we create an edge from the 'current' dump node $c.d^n$ (or the extracted dump node $N_1.d^n$, if $c = null$) to a new dump node $N_2.d^{n+k}$ representing the 'future' dump available on the receiving MCU N_2. Again, if c is a ground state component, then the edge starts at its ground state node $c.S^n$ instead of a dump node $c.d^n$. For each $dump(c, d)$ step we create an edge from c's (internal) state node $c.S^n$ (resulting from cycle $n - 1$) to the 'future' dump node $c.d^{n+k}$. In turn, for each $load(c, d)$ step we add an edge from the dump node $c.d^n$ to c's 'future' (internal) state node $c.S^{n+k}$ in cycle $n + k$. In case a ground state or implicit state of component c is modified by *load*, *inject* or *transfer*, no "aging" edges are created from the state nodes $c.S^n$ over $c.S^{n+i}$ to $c.S^{n+k}$ during the execution of the step, because the state is overridden. Finally, for each dump node d^n and each capture node r^n we add 'copy' nodes d^{n+1} and r^{n+1} at the end of each cycle and "aging" edges between them with weight *period*. Finally, note that this graph construction ignores the possibility that state objects with the same name can be created multiple times. For this analysis, reconfiguration plans must thus be transformed by renaming state dumps and capture entries so that each of them is only written once.

The resulting evolving dataflow graph can be used to **quantify the quality degradation**: The worst-case reaction time can be calculated for each cycle by taking the maximum length of the shortest paths from each sensor to each actuator in the corresponding cycle. The engineer can then see for how many consecutive cycles the reaction time would be too high (compared with the requirements of the distributed embedded application) if the reconfiguration causes delayed reaction at all. The

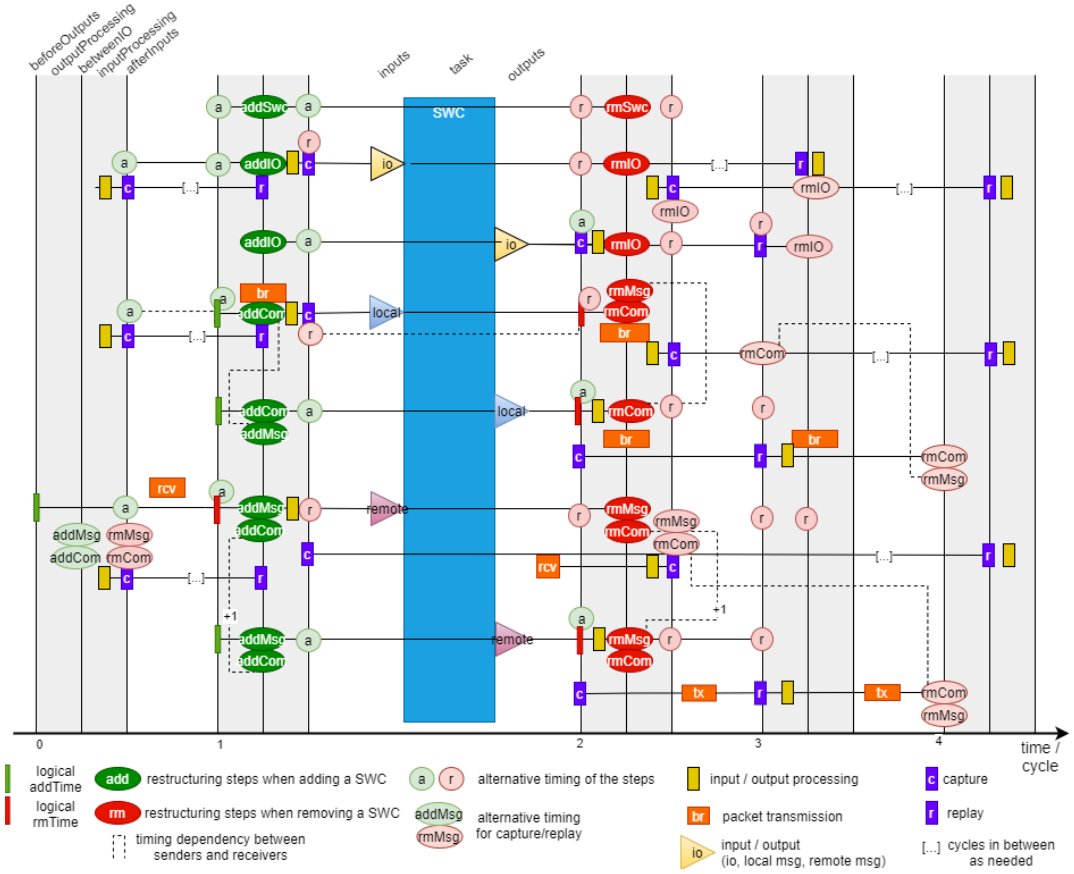

FIGURE 4.12: The general platform-independent reconfiguration timing template for restructuring and interception steps. The most straight-forward timing is to add everything in the logical cycle turnover (*betweenIO*) just before the first execution, and to remove everything in the logical cycle turnover after the last execution. However, restructuring steps can be placed at three equivalent reconfiguration hooks and in non-equivalent hooks if required for a specific reconfiguration.

duration of temporary blindness can be calculated from the number of consecutive cycles in which no path exists from a sensor to an actuator. The quality degradation of a reconfiguration plan could be plotted, for instance, in line charts showing the reaction time and blindness per each cycle and flow. The visualization of the evolving dataflow graph itself could look like in figure 4.15 in a simplified form. Chapter 6.1.2 includes evolving dataflow graphs and quality degradation analyses for different reconfigurations of the *onBtnSwitch* system.

4.3 Reconfiguration Blueprints

The reconfiguration model described in the earlier sections is based on rather small atomic reconfiguration steps. In general, it is a complex task to create a valid reconfiguration plan for any given scenario without any guidance or systematic approach. Thus, we provide a few reconfiguration blueprints and approaches to create reconfiguration plans for selected fundamental scenarios.

4.3.1 General Reconfiguration Timing Template

We provide a reconfiguration timing template showing how a software component can be added for exactly one cycle to then remove it again (see Figure 4.12). The template shows in which cycles and reconfiguration hooks the reconfiguration steps can be placed in the reconfiguration plan relative to the desired execution cycle to achieve a consistent reconfiguration. It can be used to create a consistent reconfiguration plan by allocating the *add** and *rm** steps to corresponding cycles and reconfiguration hooks for each addition and removal of software components, including temporary blocking. As of the reconfiguration model, there are three equivalent reconfiguration hooks for restructuring steps, which lead to the same reconfiguration effect on the distributed embedded application. For *addSwc* and *rmSwc* the three hooks of each cycle are equivalent (if no other step depends on the SWC state), as the new state becomes effective only at the end of the administration phase, when the agent triggers the activated tasks. For *addCom*, *rmCom*, *addIO*, and *rmIO*, the three hooks between each input processing phase (for inputs) or output processing phase (for outputs) are equivalent, because the addition or removal of these mappings becomes effective only in the next corresponding input or output processing phase. For *addMsg* and *rmMsg*, we would like to achieve the same equivalence, so that we can add a message just before adding a communication mapping to it, and remove a message just after removing the last communication mapping from it. The current model requires the mapping to be present at network transmission time, so as a current limitation for network messages, only the hook *beforeOutputs* works for outputs and *afterInputs* for inputs. However, this limitation does not apply in the platform-specific realization described in the subsequent Chapter 5, and we want to reduce the complexity for reconfiguration designer. Therefore, we describe the template assuming that all three hooks work equivalently.

The most straight-forward timing from reconfiguration designer perspective is to put all *add** reconfiguration steps in the logical cycle turnover (*betweenIO*) just before the cycle in which the component should be running for the first time (cycle 1 here). Of course, the component needs to be started before, which is not covered in the template, as well as any required coordination steps and handover steps. Similarly, it is most straight-forward from reconfiguration designer perspective to put all *rm** reconfiguration steps in the logical cycle turnover just after the last cycle of the component (cycle 2 here). This preferred timing is thus highlighted in the template. However, it may not be straight-forward to support this in the platform-specific network stack. For example, remote inputs would arrive throughout the previous cycle (see the orange rcv box), before we would schedule *addMsg* and *addCom* by the agent. Still, the packet must be accepted even though no message information has yet been created for it. After adding the message and mapping and after input processing (the yellow box), in case only afterwards packets from the previous cycle are further processed, the message and mapping must still be considered valid for the packet, even though the packet's timestamp is earlier than when the message and mapping were created. Therefore, the platform-specific model must implement a logical add time, which is not 'now' but the logical start time of the desired cycle 0 (indicated by the green bar on the left). The same three-equivalent-hook model must be applied to removal, too. For the remote input example, the remove operation is called only during cycle turnover 2, but must lead to dropping of any inputs for that now-obsolete message and mapping after the logical removal time of cycle 1 as indicated by the red bar. If, instead, *rmMsg* is scheduled *afterInputs* in cycle 1, depending on the platform-specific realization, the message information might still be kept in a

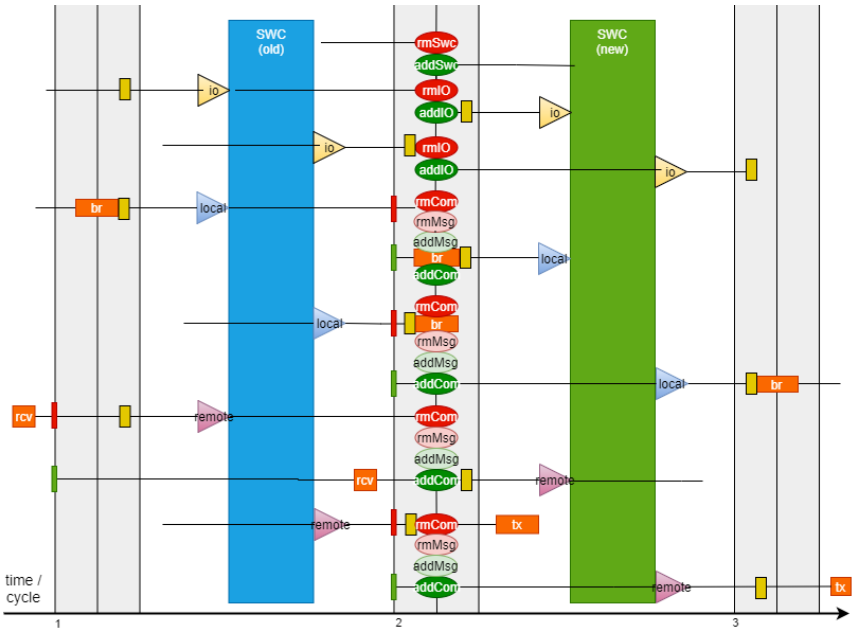

FIGURE 4.13: Reconfiguration blueprint for updating a single software component to a compatible new version using the "straight-forward" timing from the timing template: Both the *rm** and *add** steps are placed in the turnover between the last active cycle of the old version and the first active cycle of the new version.

removed state until cycle 2, so that it will exist at transmission time.

A few additional constraints are indicated in the template. Network messages and corresponding communication mappings must be added and removed so that the cycles on sender side are at least one cycle earlier than on receiver side due to the transmission delay (exactly two cycles if the preferred timing is used). This is elaborated more in section 4.3.3 on updating DAG-style applications. For local messages, both sender and receiver are mapped to the same message, which is thus only created and removed once for both. In case *capture/replay* is used, the corresponding message and communication mapping must be in place. The template shows the possible instants at which these steps could be used relative to that one-cycle lifetime of the software component. Capturing of inputs could be done in an earlier cycle, e.g. cycle 0, to then replay the input just before running the task. This might be necessary if the input provider is blocked temporarily or already removed. For the same reasons, it might also be necessary to capture the input which is processed in the single cycle for replaying it later. It might even be necessary to capture the input which would be processed just after deactivating the component, e.g. in case this component is blocked for some reconfiguration consistency reason. Outputs are captured only after active cycles of the component, and can be replayed later after removing the component. For these options, the messages and mappings need to be added and removed at different times as indicated in the template, too.

4.3.2 Minor Component Updates

Figure 4.13 shows a blueprint for the most basic reconfiguration: A compatible update of a single software component without state transfer. In such a case, no time consuming handover steps are required and no effects on other software components need to be considered. Thus, the *rm** and *add** steps can be allocated to a single

cycle turnover to replace the software component and all it's communication relationships according to the timing template. The agent will trigger the new version instead of the old version starting from cycle 2 in the figure, due to the *rmSwc* and *addSwc* steps. For I/O-mapped inputs, the agent will still apply the old mapping during input processing in cycle 1 and apply the new mapping in the input processing phase of cycle 2, just after rewiring with *rmIO* and *addIO*. The I/O-mapped outputs of the old version are still valid during output processing in cycle 2, as they are replaced only afterwards. The new mappings become effective in the output processing phase of cycle 3. Input properties mapped to local messages are handed to the old container during input processing in cycle 1. The orange *br* boxes show the corresponding *bridge* messages from the Maude model, which model the packets while logically in transit between co-located containers. Before the input processing phase of cycle 2, the communication mapping is changed by *rmCom* and *addCom* steps. If the local messages need to be changed (e.g. quota changes), *rmMsg* and *addMsg* steps must be used at the same time, in addition to reconfiguring the communication partner, too. During the subsequent input processing phase in cycle 2, the local message outputs from the previous cycle (or replayed messages) are now handed to the new version as input. For local outputs, the rewiring takes place just after the last output processing phase for the old version. The outputs are thus still available as *bridge* message for co-located receivers during input processing in cycle 2. The first outputs of the new version are produced during cycle 2 and are processed according to the new mappings in the output processing phase of cycle 3. Network-mapped properties are rewired in the same cycle turnover, according to the timing template. For remote input messages, the *rcv* Maude message received during cycle 0 is handed to the old version as input during input processing phase in cycle 1. After rewiring, the message received during cycle 1 is handed to the new version in the input processing phase of cycle 2, instead. Finally, the output properties of the old version, which are mapped to network messages, are released during the output processing phase in cycle 2 for the last time. They are transmitted to the receivers' nodes during cycle 2 (in case the message is reconfigured, note the current timing limitations of the model explained in section 4.3.1). After rewiring, the output mapping of the new version is first processed during output processing in cycle 3 and sent to the remote receivers.

Thus, it is a consistent reconfiguration which replaces the software component without changing the structure of the dataflow graph. However, many steps are allocated to one cycle turnover, if there are many inputs and outputs. As reconfiguration steps take some time (modeled by their WCET, e.g. 1 ms), it might be necessary to move some of the steps to some of the equivalent reconfiguration hooks in another cycle so that the agent does not exceed its maximum execution time (indicated by the grey areas). If the state of the old version must be transferred to the new version, the blueprint can still be used if we can achieve the state transfer under *ZET* assumption. This is the case for ground-state components, the ground states of which are fully compatible, and small enough, so that we can use *extract* and *inject* or *transfer* in addition to the restructuring steps during the administration phase in cycle 2. See INDIN-1 in section 6.1.2 for an application example in the *onBtnSwitch* system. The blueprint can not be used without adaptations if the state is too large and needs to be transferred in background, or if the components only have an implicit state which needs to be processed with *dump/load* in addition. In this case, a gap between the *rm** and *add** steps must be left, in which none of the two versions is active, and in which the handover steps are performed. For some cases, these gaps can be compensated or even reduced by further, more complex reconfiguration means (see section 6.1).

4.3.3 Updating DAG-Style Applications

We describe an approach to consistently update multiple components in distributed embedded applications, which have a directed acyclic dataflow graph between the components. This approach can be applied if the structure of the dataflow graph is preserved, i.e., there exists an isomorphism between the graphs before and after the update, where the weights of the mapped edges are equal. This is the case if all components are replaced on the same nodes along with their communication relationships over the same paths. Minor component updates should be done in independent synchronous blocks of reconfiguration steps, if possible, as described in section 4.3.2. If the application structure remains, but the message structure or semantics change in a non-backwards-compatible way, those communication partners have to be replaced synchronously. Thus, the first step is to identify the disjoint subsets of components, which have to be updated synchronously, based on compatibility-breaking interface changes. For instance, if an output type at a component is changed from an integer to a floating-point number, and the receiving component's input type is adapted to this change, then these two components must be updated synchronously. Obviously, these subsets must be merged transitively, so that if components c_1 and c_2 must be replaced synchronously, and c_2 and c_3 must be replaced synchronously, too, then all three components must belong to the same synchronous subset $\{c_1, c_2, c_3\}$. In the extreme cases, all components of a distributed embedded application must be replaced synchronously, or all components can be replaced independently. For each identified subset, we create a synchronous block of reconfiguration steps in the reconfiguration plans for the corresponding nodes. Then we plan restructuring steps so that the components are replaced consistently with the dataflow timing. The replacement for each component c_i is initialized according to the minor update blueprint: We put all the *rm** steps for removing the old version and all the *add** steps for adding the new version in the hook *betweenIO* in one cycle n_{c_i} (equivalent hooks may be chosen for the individual steps at the end, if needed). Following constraints exist for the replacement offsets n_{c_i} to consistently maintain communication relationships during the update:

***local messages*:** the replacement cycle n_s for the sending component s must be one cycle before replacing the receiving component r in cycle n_r.

***network messages*:** the replacement cycle n_s for the sending component s must be two cycles before replacing the receiving component r in cycle n_r (or: one more cycle for each additional transmission cycle in case of multiple hops)

To assign replacement offsets to each component, choose a component c_i, which has no message inputs in the subgraph, and assign an initial offset $n_{c_i} = 0$. Then for each receiver c_j of a message from the selected node, assign $n_{c_j} = n_{c_i} + 1$ in case of local messages, and $n_{c_j} = n_{c_i} + 2$ in case of network messages (or in general, $n_{c_j} = n_{c_i} + 1 + k$ for k transmission cycles). Continue with one of the nodes for which an offset was assigned and which has unhandled receivers or senders. For its receivers, assign the offsets in the same way. For its senders, assign offsets in the opposite way, i.e., by reducing the offset by $k + 1$ for k transmission cycles. This can be repeated until all components in the weakly connected subgraph have offsets assigned. Finally, if n is the smallest offset assigned, we add $|n|$ to all offsets n_{c_i} so that the first replacements are planned for offset 0 after the synchronization point (this fixes the case $n < 0$). Figure 4.14 shows a blueprint for a synchronous reconfiguration of three software components *SWC1*, *SWC2*, and *SWC3*. The components *SWC1* and *SWC2* are co-located on *node1* and there is a sender-receiver relationship

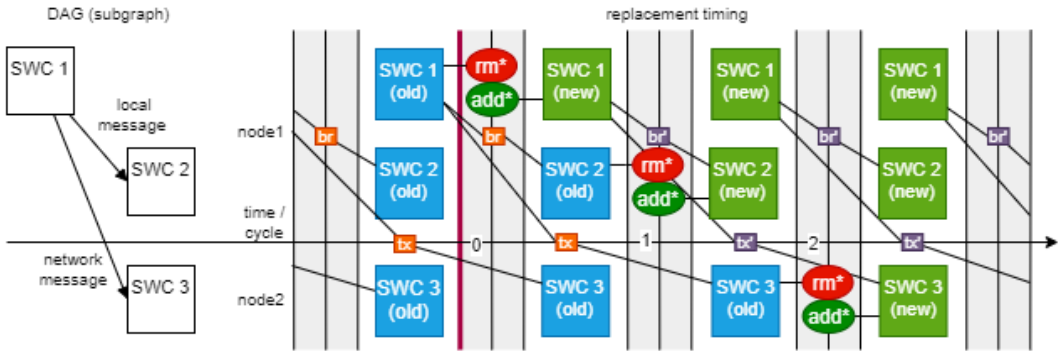

FIGURE 4.14: Reconfiguration blueprint for updating a DAG-style application. The individual component replacements are done with the minor update blueprint. The offsets of the replacements are calculated from the communication graph.

SWC1 → *SWC2*. Component *SWC3* is located on *node2* and there is a sender-receiver relationship *SWC1* → *SWC3*. The three components can be updated consistently as follows, if only the corresponding old and new versions are compatible. According to the rules of the reconfiguration approach, *SWC1* is replaced at offset 0 from the synchronization point. *SWC2* is replaced one cycle later in cycle 1 due to the local message and *SWC3* two cycles later due to the network message. Effectively, the outputs of the old version of *SWC1* are processed by the old versions of *SWC2* and *SWC3* due to the deterministic communication timing. The outputs of the new version of *SWC1* are processed by the new versions of *SWC2* and *SWC3*. As other communication relationships beyond the synchronous subgraph remain compatible, they work with any of the versions of *SWC1*, *SWC2*, and *SWC3*.

This approach guarantees a consistent update without quality degradation for stateless DAG-style distributed embedded applications. If state transfer is required and the corresponding components have ground state kind, then the same timing can be used with additional handover steps *extract*, *inject* and *transfer*, if they can be used under *ZET* assumption. The timing of the update must be adapted if the state becomes too large for this (as of the analysis described in section 4.2). Adaptation is also needed, if one of the steps *transmit*, *dump* or *load* is required or if a state must be transformed by an update package component. As semi-systematic approach for this, we propose to start with a reconfiguration plan as of this blueprint and add the handover steps, which can be done under *ZET* assumption. Afterwards, the remaining handover steps are added one after another, starting with the first replaced components, shifting the replacement offsets of components "behind" according to the caused delay in the dataflow. When replacements are shifted, inputs are missed and outputs may be missing at other components, so the shifts must be handled either by blocking or by patching the broken dataflow with *capture/replay* or with temporary update package components. In such cases, the impact on other software components beyond the selected subgraph must be considered, too. Figure 4.15 shows an example, in which the state of *SWC1* needs to be dumped, while the remaining steps to transfer the state to the new versions can be done in the cycle turnovers before activating the new versions. We calculate the number of cycles $k = \lceil eta/quota \rceil$ needed for the dump operation according to the analysis. We then insert this step into the reconfiguration plan and shift the replacement cycles of components "behind" *SWC1* by k, recursively. In this case, if *dump* needs one cycle ($k = 1$), then the new replacement offset for *SWC2* is cycle 2, and cycle 3 for *SWC3*. Three *transfer* steps are used during the replacement cycles under *ZET* assumption. As *SWC1* is blocked while

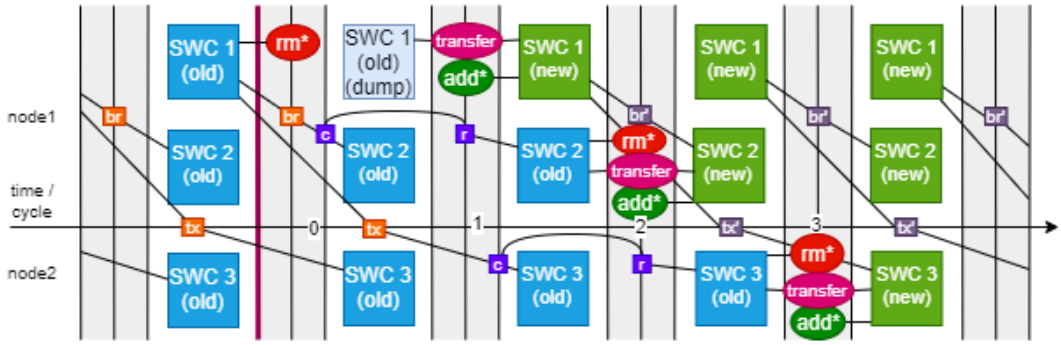

FIGURE 4.15: Example for an adaptation of the blueprint to transfer the states: due to the time-consuming *dump* step at *SWC1*, the replacement offsets of *SWC2* and *SWC3* are adapted and the broken dataflow is patched with *capture* and *replay*.

running the *dump* step, no output is produced and no input is available in that cycle for *SWC2* and *SWC3*. This broken dataflow is patched with *capture* and *replay* steps in this example. Note that all components may have additional communication relationships, which must be considered during reconfiguration design, if any quality-degrading adaptations of the blueprint are required.

4.3.4 Application Start and Stop

The initial start and the final stop of a distributed embedded application are special cases of dynamic reconfigurations. When starting the application, it would be possible that all nodes start their local application containers and then activate them in the same cycle. This way, however, inputs from other components would not be available in the first cycle. One way to handle this is to implement the component in a way which can work around temporarily missing inputs. This might be a good idea from a robustness perspective, but if this was our general approach for starting, it would mean that we mandatorily require reconfiguration-related logic within the component implementation, violating the desired separation of concerns (functionality vs. reconfiguration). Additionally, input-triggered tasks would not be possible in this approach. Another way would be to provide predefined initial inputs from outside via *replay* steps. In this approach, we would need an application-specific input, while we want to keep the message contents transparent to the platform. This could be provided by a captured output from the engineering environment, or could be captured in an extended container start phase. At the moment, this is not supported in our approach, and it would also add more complexity and component interface requirements.

We propose to start a distributed embedded application in the same way in which it would be updated, only without removing old versions. For DAG-style components, this can be done according to the blueprint described in section 4.3.3. In the example shown in figure 4.14, after starting and synchronizing before, we would add component *SWC1* in cycle 0, *SWC2* in cycle 1, and *SWC3* in cycle 2. Using this approach, no extra adaptation is needed, and all components get valid inputs from their first cycles on. The I/O-mapped outputs of the components will not be provided from the first cycle on, though. As the distributed embedded application is just starting and has not provided a single output before, it should be acceptable. The outputs must have been initialized and maintained safely during system start before activating the components, anyways. If the distributed embedded application has any cycles in the communication relationships, then this approach can not

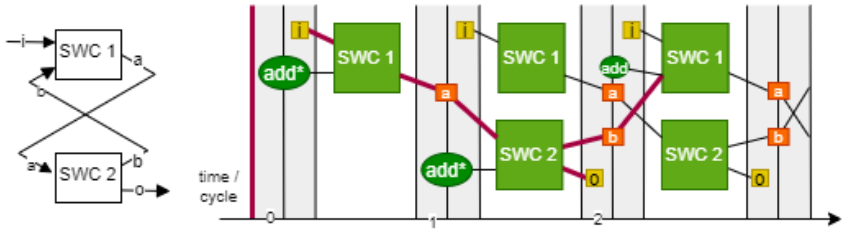

FIGURE 4.16: Example for starting a non-DAG application. Two co-located components *SWC1* and *SWC2* with circular communication can be started with the DAG-timing, if *SWC1* can temporarily work without the non-arriving input *b*. The purple lines show the walk through the graph leading to this timing.

be used in this pure way. We can still solve the cycle by considering which component is robust against which missing inputs and for which we can provide initial inputs from outside. If we initially ignore communication relationships for such inputs, because their non-arrival is acceptable, the distributed embedded application can be transformed to a DAG-style application. For the resulting application we can define the start sequence as described before, potentially providing outputs from outside where needed. The ignored communication mapping needs to be added according to the dataflow. To derive the exact offsets for adding the mappings, we can think of the application as if the components with missing inputs were split, so, the part receiving the real input would be added later. Or more precisely, we seek for a parallel walk through the communication graph, in which we take each edge (message) exactly once, and all outgoing edges of a visited node (component) at the same time, so that only ignorable incoming edges are taken later. Each edge leads to an offset increase of 1 for local messages and n for network messages with n transmission cycles. When we first visit a component, it is activated and all non-ignored communication mappings are added at the current offset. If inputs need to be provided by the agent, the mapping is added at the same time and *replay* is used. Otherwise, the communication mappings are added only when the corresponding edge is used. Figure 4.16 shows an example for such a non-DAG application start. State transfer is not needed in this sequence, as any initial state is loaded during container start.

For stopping an application, the same approach could be used in general. However, if the application does not need to perform any stop control logic, we can just stop the system at a specific time, which is the current approach. If some stop control logic is needed, the current model does not provide a means to align the timing of the stop procedure within the application and the stop reconfiguration. In future work, some degree of self-reconfigurability could be introduced to enable a stop interaction between the application and the agent.

Chapter 5

Evaluation Platform: Real-Time Container Architecture

We describe a realization of the runtime platform concepts modeled in chapter 3 and chapter 4: The real-time container architecture. The architecture provides a reference solution, which executes and reconfigures distributed embedded applications accordingly. Its main purpose is to show that the platform concepts are feasible, especially with regard to the underlying goals and assumptions. We developed a prototypical implementation to realize concrete control systems for demonstration and evaluation purposes (see chapter 6). The architecture description provided by this chapter is needed to understand the evaluation setup and results. In earlier publications, we already described some parts of the architecture in a short and abstract way. [TSK18] provided an overview of the technology stack and the main routine without reconfiguration. [TK19a] described the initial reconfiguration extensions (without state transfer) and first measurements based on an older prototype. [TK19b] added the pieces related to state transfer. However, due to the strict format requirements these publications had to fall short of a substantial evaluation platform description. In this chapter, we therefore provide architectural details on our prototype as far as they contribute to the understanding of the evaluation.

The structure of this chapter and the structure-related glue phrases are based on the arc42 template [SH16] for light-weight architecture documentation. We first give a compact overview of the platform aims (goals, requirements, and scope). Then we describe the structural and behavioral concepts of the real-time container architecture from different angles. Finally, we outline relevant aspects of complementing systems around the distributed control system, which are needed for its development and operation.

5.1 Platform Overview

We briefly describe important problem-space aspects of the real-time container architecture: Functional and non-functional requirements and scoping.

5.1.1 Introduction and Goals

The real-time container architecture prototype is a runtime platform for distributed embedded applications, which calculate outputs from inputs in isochronuous tasks (see figure 5.1). Its main purpose is to demonstrate and evaluate the platform concepts developed within the scope of this thesis. Due to the limited scope, the current architecture is only an initial break-through version, which does not address all industrial requirements (e.g. security, compatibility to certain standards, ...).

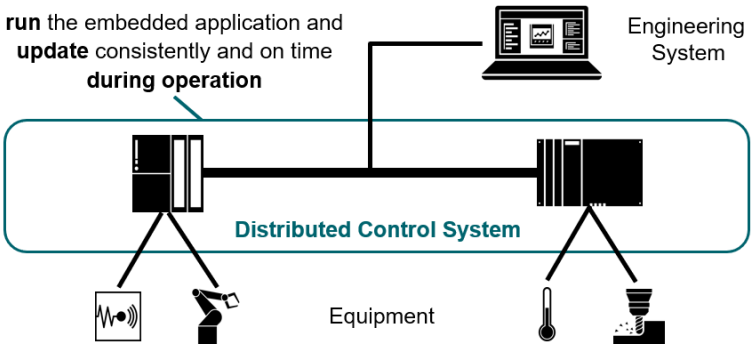

FIGURE 5.1: High-level system overview showing the purpose of the real-time container architecture. Using our runtime platform, modular and distributed embedded applications control the installed equipment consistently across the nodes. The software components can be updated during operation by means of reconfiguration plans. Logging and tracing features are provided for testing and evaluation.

However, the architecture addresses the substantial requirements of current and future industrial control systems targeted by this thesis (especially isolation, temporal determinism, and updating during operation). We have designed the platform concepts and the concrete real-time container architecture with the goal that future work can address more requirements and make the platform suitable for more and more classes of control systems.

Requirements Overview

Table 5.1 shows an overview of the functional requirements from the end user's point of view. The main requirement is the possibility to setup the nodes of a distributed control system and then configure and run distributed embedded applications (eApps) on these nodes. For developing, demonstrating and evaluating the real-time system it is required that the operator can retrace its functional and temporal behavior by means of logging and runtime measurements.

ID	Requirement	Short Description
M-1	Setup nodes	Install and configure the platform (including the OS) according to the network topology.
M-2	Configure eApp before operation	Describe an eApp so that the platform can configure and execute it accordingly. Each node is provided with a local view on the deployment (extended by O-4).
M-3	Start and stop eApp	Provide a simple method to start and stop an eApp simultaneously accross nodes. Components may be pre-installed (extended by O-1).
M-4	Embedded control by eApp	Run embedded functionality given by an eApp, so that orchestrated application-layer components can control the equipment of a distributed system.
M-5	Reconfigure eApp during operation	Perform dynamic reconfigurations based on reconfiguration plans as described in chapter 4. The plans and components may be pre-installed (extended by O-1 and O-3).

ID	Requirement	Short Description
M-6	Logging of platform and eApp execution	Logs of the platform and eApps are accessible at least after execution. Log levels are configurable before start time.
M-7	Measure and report real-time behavior	After execution of an eApp, measured CPU time statistics (min, max, average) are available for the cyclic administration routine, reconfiguration steps and tasks, as well as drift and slack times.
O-1	Automatically (pre-)install components	Components can be pre-installed within M-2 and M-5 based on an eApp configuration and/or a reconfiguration plan. Otherwise, they are downloaded dynamically by download steps.
O-2	Measure resource consumption	The platform captures usage (min, max, average per cycle) of CPU, memory and network by platform- and application-layer components.
O-3	Dynamically roll-out reconfiguration plan	Reconfiguration plans can be rolled-out during eApp runtime. The plans are parsed in background and then executed by the platform.
O-4	Overall deployment description.	The complete distributed system can be configured automatically for a specific eApp on a specific topology from an overall deployment description.

TABLE 5.1: Functional requirements of the real-time container architecture. From a user's point of view the platform enables operation of a distributed control system. M-* features are mandatory to enable the evaluation. The optional O-* features provide significant improvements, e.g. in engineering efficieny and presentability.

Quality Goals

Table 5.2 shows the most essential quality goals for this evaluation platform. The platform concepts have been defined to improve modularity, adaptability, reusability and engineering efficiency of distributed control systems. However, this section focuses on concrete quality goals for the real-time container architecture itself. They result from the top-level goal to evaluate the platform concepts defined in chapter 3 and chapter 4 regarding their suitability for the requirements of real control systems.

ID	Quality	Short Description
Q-1	Performance	The evaluation platform must support eApp periods of 100 ms (regardless of reconfigurations). Periods of 50 ms, 20 ms and even 10 ms are desired. The long-term goal is 1 ms. The platform overhead should be lower than 10 % regarding CPU, RAM and network consumption per cycle.
Q-2	Time behavior	The clocks and cycles must be in-sync on each node with less than 1 ms drift for the complete runtime. All inputs, outputs, tasks, messages and reconfiguration steps are triggered and completed according to the defined timing (see chapters 3 and 4).

ID	Quality	Short Description
Q-3	Isolation	The software components shall be technically isolated, so that they can only interact with their environment in ways explicitly enabled (ports, APIs, ...). It must not be possible for a component to select or detect the communication partner. The communication partners must be transparent to a component – also during reconfigurations. Shared resources must be restricted to the declared demand (CPU quota: *wcet/period*, network quota: *msgsize/bpp*).
Q-4	Portability	The platform must be largely agnostic about the technology stack used by software components. Input/output handling and task handling should be based on standard POSIX features (e.g. sockets, signals, ...) so that components can use arbitrary programming platforms (C/C++, ...) – at least it should be easy to add support for a platform. Porting the platform to additional MCU types should require minimal modifications.

TABLE 5.2: The most essential quality goals of the real-time container architecture.

Stakeholders

This description of the real-time container architecture primarily aims at researchers from academia and industry, who wish to better understand the platform concepts, key design decisions, and the evaluation results presented in this thesis. As part of a thesis it is not meant as material to convince product managers or customers to use or further develop the platform. Even though we give a comprehensive overview of the architecture, this is also not a manual for platform developers, users of the evaluation prototype, or architects of other systems which may want to integrate or support this platform in future.

5.1.2 Architecture Constraints

When developing a platform in a product context, there are usually many organisational and technical constraints, which limit the architectural freedom. As the platform concepts described in this thesis are rather novel and to some extent unconventional, we had to conduct this research largely decoupled from product development. This opened up the design space, but lead to the constraints shown in table 5.3.

ID	Short Description
C-1	An independent prototype is developed instead of working on a product.
C-2	The concepts and evaluation results are published as part of academic work, even though it is done in collaboration with Siemens.
C-3	The platform should be built on/for open products – favorably by Siemens. This is a consequence from C-1 and C-2.

TABLE 5.3: Architecture constraints limiting the design space.

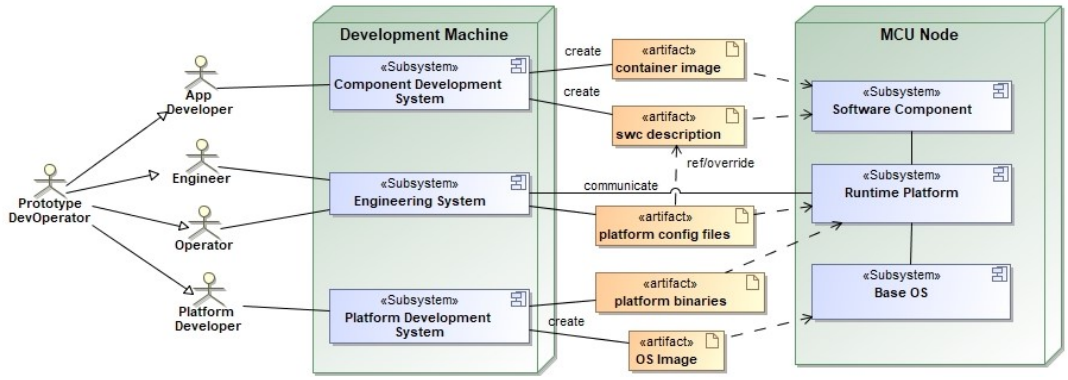

FIGURE 5.2: High-level system overview showing the scope of the real-time container architecture prototype. The 'full-stack' DevOperator creates and rolls out different runtime artifacts and commands to define and run a prototypical distributed control system.

5.1.3 System Scope and Context

An industrial control system and the runtime platform in particular are surrounded by multiple other tools, systems and use cases. This includes tools for development and monitoring, engineering and operation, as well as other control and supervisory control systems. There are industrial standards for each of these interfaces, but also proprietary and legacy solutions. For the scope of this thesis, we do not consider compatibility with any specific neighboring system. Instead, we define and integrate neighboring systems as needed (e.g. a development environment, a deployment mechanism, ...). Regarding the communication with sensors and actuators, we only consider the equipment used during the evaluation (e.g. GPIO-based field-level control). Thus, there are no obligations by neighboring systems.

Figure 5.2 shows an overview of the evaluation platform's scope and context. The prototype DevOperator acts as app developer, system engineer, platform developer and prototype operator. Using suitable component development systems, the app developer creates executable software components along with software component descriptions. The components are provided in a largely transparent bundle, the container image. The platform developer uses the platform development system to create the runtime platform binaries and operating system binaries (in this case an OS image). The engineering system is used by the engineer to re-/configure the platform for a specific deployment (see section 3.1.4) of the distributed control system. It is also used by the operator to start, stop, and inspect the runtime prototype. For these purposes, the runtime platform running on each MCU node must communicate with the engineering system. During operation, the platform components on the different nodes setup and run the distributed control system in coordination with each other. As part of this they manage the lifecycles of application-layer components and execute reconfigurations based on given reconfiguration plans.

5.2 Runtime Platform Architecture

We describe the real-time container architecture from the solution-space point of view. After an overview of the underlying solution concepts, we describe the building blocks of the real-time container architecture as well as their interactions and their deployment. Then we elaborate on cross-cutting concerns such as the process

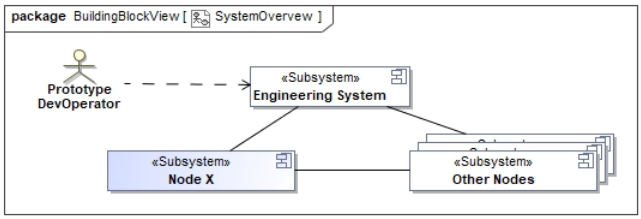

FIGURE 5.3: System overview showing the high-level subsystems of the real-time container architecture prototype. In this section we focus on the details of *Node X*, which represents one MCU in the distributed control system.

control, reconfiguration, communication, logging, and building/deployment. This section focuses on the runtime platform (cf. *Node X* in figure 5.3). The subsequent section 5.3 describes complementing subsystems on the engineering system.

5.2.1 Solution Strategy

For the design and implementation of the evaluation platform we made following fundamental decisions:

Target System: We use SIMATIC IOT2040 devices as target systems and build an OS image with minimal customizations based on the example image available online [Sie18]. The example image is open and customizable and the yocto build can easily be run in a build container using the kas tool. The IOT2040 is an edge device, i.e., it has enough resources to run such a modern platform while it is still comparable to field devices in industry.

Container Runtime: We use Linux containers (LXC) [Can22] as app execution environment on the target systems – one container per software component. The components are deployed in individual, self-contained provisioning packages for functional and non-functional isolation and to decouple their lifecycles. At the time of this decision, LXC was the best choice regarding maturity, openness, API access, system commit, availability on the target system and experience within the team. Other OCI runtimes without interfering daemons should work as well (e.g. runc).

Orchestrator / Daemon (Agent): On each target system a custom real-time container agent (agent) is employed (see section 5.2.2). It re/configures the system and the application containers in coordination with each other and with the engineering system according to our runtime concept. The agent is implemented with C++ for best performance of the administration phase – especially of reconfiguration operations. A background worker component is running in a dedicated cgroup for resource control, but not within the more isolated LXC containers. We do not use the available container daemons and orchestrators (LXD, Docker, Kubernetes, ...), because they do not support our runtime concepts for dynamic orchestration with end-to-end determinism (especially the LET paradigm, the reconfiguration concept, and the I/O handling). Some aspects like monitoring and container lifecycle management (excluding the synchronous reconfiguration) could be managed from those existing tools in future work, though.

Scheduling: Four logical priorities are distinguished by our prototype: minor, normal, elevated, critical. The cyclic administration phase is executed by a dedicated process scheduled with the policy SCHED_DEADLINE to ensure isochronuous invocation at highest priority (critical). The kernel should be patched and

configured for a fully preemptive kernel (PREEMPT_RT_FULL) to achive minimal scheduling latency (planned). To run the containerized software components at SCHED_FIFO, the kernel needs to be configured with RT_GROUP_SCHED enabled. All application-layer components run at the same priority (normal) and share the resources (CPU, memory, network) according to an agent-configured quota per cycle. The cyclic task execution is aligned with the isochronuous tick using a sleep/signal approach: The components wait for a SIGINT signal before starting the next cycle, which the agent sends to the active components at the end of the administration phase. This simplifies the scheduling requirements, but has to be done by the components, actively. The background worker is scheduled like an application-layer component (normal). The agent's listener threads run elevated at SCHED_FIFO with a higher priority than application-layer components and lower than the cyclic administration phase. Supporting processes such as remote shells are scheduled at minor priority (SCHED_NORMAL). If needed, they may be bandwidth-constrained like the background worker.

Clock/Cycle Synchronization: Clock synchronization is done via Precision Time Protocol (PTP), which is running on each node at highest priority. Additionally, the agents align their isochronous administration phases and cycle counters with a master agent as described in section 5.2.3.

Communication and I/O Mappings: Component communication is possible via UDP or files (see section 5.2.3). The agent uses firewall techniques such as Network Address Translation (NAT) to create the communication relationships for UDP. NAT, network bandwidth control and alignment with the cycles are achived via a custom 'barrier' Qdisc. The barrier Qdisc keeps back messages until they are released by the agent and enough quota is available. For the file-based communication, the agent reads or writes specific files within the root file system of the components' containers during the administration phase in order to move the contents according to the communication- and I/O-mapping.

Reconfiguration Steps: Reconfiguration steps are implemented according to the visitor pattern, which makes the agent open for new step kinds (see section 5.2.3). The reconfiguration steps provide functions which modify the data structures and the system state accordingly using the agent's interface (see section 5.2.5).

Real-time Measurement: For measuring specific execution times with minimal overhead, a custom stopwatch is built into the agent (see section 5.2.5).

Logging: Logging is configurable via rsyslog and via agent options (see section 5.2.5). There are three setups: 1) For achieving the best performance, we turn off rsyslog and any logging (eval). 2) For better insights of the time-critical system behavior during development we turn on rsyslog with a local-only configuration and selected log levels (dev). 3) For collecting and plotting the logs (incuding runtime statistics) on the engineering system during operation, we additionally configure rsyslog for log forwarding (demo).

Re/Configuration Files: The agent uses Yaml-based configuration files and reconfiguration plans given in a text-based RcPlan DSL (see section 5.2.5). This DSL looks similar to the constructor functions used in chapter 4. The plans are converted to reconfiguration step objects by the background worker using an ANTLR-based parser.

Resource Monitoring: For resource monitoring during development, additional optional components are used as needed, e.g. a cadvisor container and top.

Setup and Installation: Node setup, installation of the platform and pre-installation of application-layer components are done by dedicated scripts. For now, the container images and configuration files need to be created and deployed consistently with the software component and deployment descriptions (see section 5.2.5).

5.2.2 Building Block View

We describe the static decomposition of the system on different levels of abstraction across the hierarchy along with the important relationships.

MCU Node Overview

Figure 5.4 shows a whitebox overview of the architecture on one MCU. Table 5.4 gives an overview of the depicted building blocks. We use containerization as much as possible to achieve the desired functional and non-functional isolation. Platform components are trusted and need leveraged access to the system, so they are put into privileged containers (e.g. they have full visibility of other components). Application-layer components only get the access they need (UDP ports, I/Os and files). The real-time container agent (agent) is the central runtime platform component. It runs on each node and orchestrates the distributed embedded application using the barrier component. The barrier hooks deep into the network stack to compatibly offer re/configurable firewall features: NAT, bandwidth control, capture/replay, and most importantly a certain traffic shaping functionality. Using the barrier's traffic shaping functionality, the agent can align the transmission of network packets with the isochronuous period of the distributed embedded application. To also align the I/O access with the period, the agent actively conveys the inputs and outputs between the containers and the drivers during the cycle turnover. The isochronuous cycle alignment of the agents across nodes depends on the clocks, which are synchronized via PTP using ptp4l [Coc20]. The node and the agent can be administered from the engineering system based on SSH [YCB+13] (containerization planned). A containerized background worker component on each node executes reconfiguration steps in background as requested by the agent. The containerization of these platform components helps to monitor and re/configure the distributed control system without interfering with the embedded application. For the same reason, the platform-layer components and their interfaces with the agent are designed with the primary purpose to keep time-consuming and unplanned activities away from the agent as much as possible.

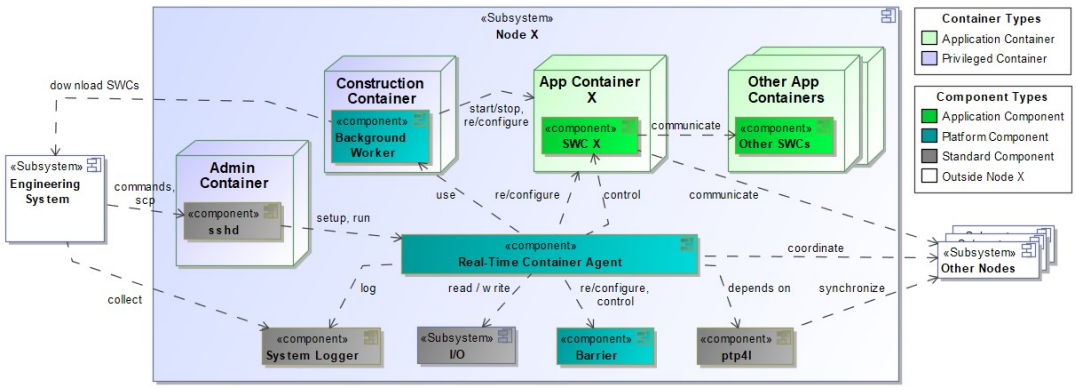

FIGURE 5.4: The components involved on one MCU in the real-time container architecture prototype (see table 5.4).

Name	Responsibility
SWCs	Cyclically compute outputs from their inputs and state.
Real-Time Container Agent	Cyclically triggers the SWCs' tasks and handles their inputs and outputs. Re/configures the node-local components in coordination with agents on other nodes.
Background Worker	Performs reconfiguration steps in background, e.g. download a container image (planned), start a container, ...
System Logger	Configurably handles log messages from the other components providing high-resolution timestamps (ns).
I/O	Drivers for read/write access to installed sensors and actuators.
Barrier	Helps the agent to enforce the logical execution time paradigm for socket communication through traffic shaping and NAT.
ptp4l	Continuously synchronizes the system clocks of the nodes.
sshd	Provides general administration access for development and operation purposes via secure shell (SSH).

TABLE 5.4: Short explanations for the components of the runtime platform architecture overview (see figure 5.4).

Real-Time Container Agent

Figure 5.5 and table 5.5 show the components of the real-time container agent running on each node. The agents re/configure the system and the application containers in coordination with each other according to our runtime concept:

- Implement the logical execution time paradigm by processing inputs and outputs and triggering tasks at the specified isochronuous instants.
- Execute the required reconfiguration steps at the right times according to a given reconfiguration plan.

A background worker component is running in an isolated cgroup (called construction container). It executes time-consuming steps in background on command of the agent. The bidirectional communication between the agent and the background worker is done asynchronously via ZeroMQ Pub/Sub [Hin13]. A dedicated listener thread of the agent receives messages from the background worker. It accepts messages and puts them into a shared memory location (the agent state). Dispatching of these messages is mostly done by the agent's cyclic thread, while it performs the cyclic administration phase. The same strategy is used for inter-agent communication, only that this (remote) communication is based on UDP. Consequently, such messages can be lost. From a robustness perspective, this is a weakness of the prototype, which must be addressed by future work (e.g. by repeating messages in the synchronization protocol). Note that TCP alone cannot be used to solve this, because the timing of inter-agent communication is crucial. As managing wrapper around the mentioned agent components, the agent main component is added. It can be launched via SSH and may launch remote agents via SSH, if this is configured in the node configuration file (see section 5.2.5). Additionally, the agent main component is responsible for starting and stopping the other local agent components accordingly. The background worker is started as a separate process using fork. The agent's cyclic component and the listener are additional threads within the main process's context. Finally, the additional barrier component is started by loading the corresponding kernel module. The agent's main component stops these

processes and threads either on command or after a configurable timeout by setting termination flags and sending cancellation signals. For re/configuration purposes, the agent components communicate with the network stack and the barrier component via netlink [SKKK03], a socket-based interface to kernel-space components. GPIOs are accessed using the file-based sysfs interface [Ker21]: The agent's cyclic component copies the file contents between the specific files within the sysfs (under `/sys`) and the application containers' root file system (rootfs). The RAM-only file system tmpfs is used within the rootfs and in host locations wherever the agent or any other component works with files in time-critical paths. To trigger a cycle of a software component, the agent sends the `SIGINT` signal to all processes within the corresponding containers' cgroups. Finally, we describe the communication of the agent with the engineering system. It is basically implemented via file uploads and downloads. Before starting the agent, the DevOperator creates the node configuration file and the initial reconfiguration plan file on the target system. These files are copied to the nodes via deployment scripts which use SCP. Additionally, container images can be pre-installed via deployment scripts, too. During operation, further reconfiguration plans can be downloaded to the target system in background. Reconfiguration plans are pushed by the engineering system via SCP, parsed by the background worker and then added to the agent's list. The same mechanism can be used behind the scenes by the update management system described in section 5.3.3. It is planned that container images can be pulled from a container registry within the construction container. Table 5.6 shows an overview of the most important interfaces of the real-time container agent.

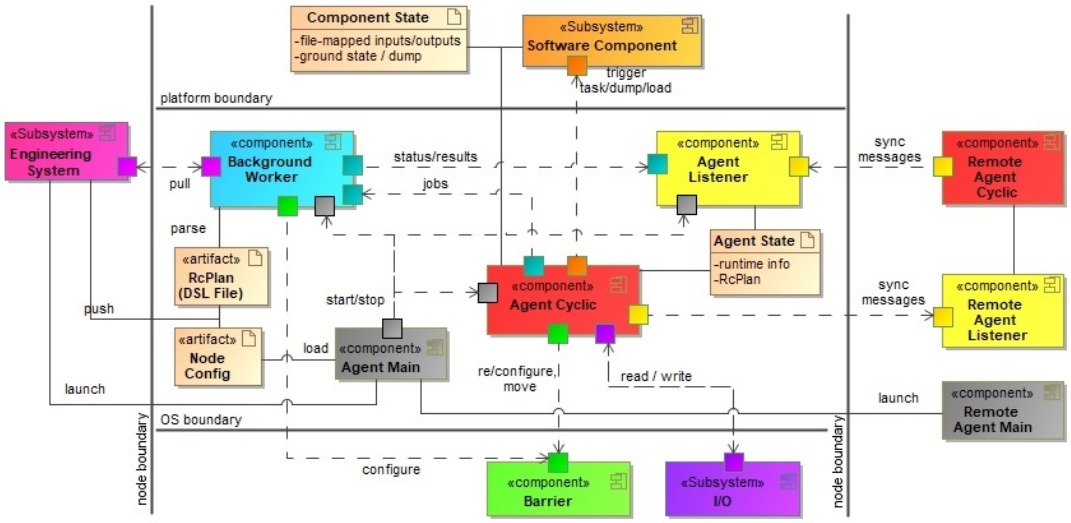

FIGURE 5.5: The components of the real-time container agent (see table 5.5).

Name	Responsibility
Agent Main	Sets up the system and the other agent components and terminates them according to the node configuration file.
Agent Cyclic	Isochronuously performs the administration phase: process outputs, process inputs, trigger tasks and perform/trigger reconfiguration steps in-between.
Agent Listener	Receives messages from the background worker and from remote agents and updates the shared agent state accordingly.

Name	Responsibility
Background Worker	Performs reconfiguration-related operations in background: start/stop components (download planned), transfer state, and parse reconfiguration plans.
Barrier	Helps the agent to enforce the logical execution time paradigm for socket communication through traffic shaping and NAT.

TABLE 5.5: The components of the real-time container agent (see figure 5.5).

Name	Relationship	How	Short Description
jobs	Cyclic → Worker	ZeroMQ	Worker subscribes to `agentpub` topic.
results	Worker → Listener	ZeroMQ	Listener subscribes to `bgworkerpub`.
sync	Cyclic → Listener	UDP	Cyclic sends to listener port `31337`.
trigger	Cyclic → SWC	signal	Cyclic sends SIGINT.
move	Cyclic → Barrier	netlink	Cyclic sends commands to the barrier module via a `NETLINK_ROUTE` socket.
io/state	Cyclic → SWC	rootfs	Cyclic reads/writes the containers' files.
gpio	Cyclic → I/O	sysfs	Cyclic reads/writes files under `/sys`.
pull	Worker → ES	undefined	Worker pulls container images (planned).
push	ES → Worker	SCP	ES pushes RcPlan DSL files via SCP.

TABLE 5.6: The most important interfaces of the agent.

5.2.3 Runtime View

We describe concrete behavior and interactions of the components described before. Not all use cases are documented here, because we only elaborate on selected, representative scenarios, which are essential for understanding the prototype.

Agent Cyclic Routine

The agent cyclically performs the administration phase to manage the system execution. In essence, the agent's duties are to enforce the logical execution time paradigm and to perform dynamic reconfigurations. We formally specified this runtime behavior in the previous chapters 3 and 4. The basic behavior (without reconfiguration) is specified in section 3.2.3 and illlustrated by the corresponding figure 3.9. In the formal specification, the agent actively extracts outputs from the application containers and handles them according to the defined mappings. Then the agent actively injects the inputs. Afterwards, the agent completes the administration phase by triggering the active components' tasks. In addition, there is a reconfiguration hook prior to each of these sub-phases of the administration phase. The behavior of the agent within the reconfiguration hooks is specified formally in section 4.1 and illustrated in corresponding figure 4.3. In the formal specification, the agent checks the current state of the reconfiguration plan and performs applicable actions as defined by the corresponding reconfiguration step kind and parameters.

Listings 5.1 and 5.2 show a pseudo code overview of how the evaluation prototype implements this behavior (cf. [TK19a]). The cyclic agent routine is executed in a dedicated thread running with scheduling policy `SCHED_DEADLINE` (see system start later in this section). This makes the cyclic agent routine the highest-priority

```
op doCycle:
  if not healthy():
    trigger fault reaction;

  rcHook(beforeOutputs);

  for each component c
    put I/O-mapped outputs of c to driver;
    move egress barrier;

  rcHook(betweenIO);

  for each component c
    get I/O-mapped inputs of c from driver;
    move ingress barrier;

  rcHook(afterInputs);

  for each component c:
    interrupt processes of c;
```

LISTING 5.1: The cyclic agent routine with three *rcHook* calls.

```
op rcHook(hook):
  if rcPlan.isEmpty()
    check new rcPlan;
    return;

  while !rcPlan.isEmpty()
    RcStep step = rcPlan.front();

    if step.isDone() // e.g. in background
      continue with next step;

     // Visitor pattern
    if step.doesApply(Agent.this, hook)
      step.apply(Agent.this);

    if step.isDone()
      continue with next step;
    else
      break; // e.g. continue later
```

LISTING 5.2: The reconfiguration hook operation.

workload in the system and activates it at configurable isochronuous instants. Additionally, these instants are aligned across nodes due to the system start routine and continuous clock synchronization via PTP. All agents also have the same cycle counter throughout the complete system lifetime. When activated by the scheduler, the *doCycle* operation of the agent is called (see listing 5.1). The agent first checks the **health** of the distributed embedded application by checking specific files `_start` and `_end` within the container of each activated software component. The components append the current timestamps to these files when starting and completing each cycle. The agent uses these timestamps to gather runtime statistics via the stopwatch component, but also to detect whether the components successfully complete their tasks. Currently, this is only a monitoring feature and a placeholder for more mature health checks and fault reactions to be enabled in future. After the health check, the **first reconfiguration hook** is reached, which is implemented by the *rcHook* function (see listing 5.2). In this hook, the agent may take additional actions at corresponding times based on the reconfiguration plan. As indicated by the pseudo code, the agent checks whether a new reconfiguration plan is available whenever the current reconfiguration plan is empty. If this is the case, the new reconfiguration plan is parsed in background as a supporting pseudo reconfiguration step. In a subsequent cycle, the new plan is available as list of parsed *RcStep* objects and replaces the empty list. Each *RcStep* kind specified formally in section 4.1 is implemented in a dedicated class derived from *RcStep* in the same hierarchy as modeled in Maude (see figure 4.1). The agent behavior is extended by each step kind according to the Visitor pattern: The steps implement two functions *doesApply* and *apply*, which both take the agent as one of the parameters. The *doesApply* function defines the timing (e.g. returns true if the hook and cycle match the offset of a modification step) and the *apply* function performs the step-specific reconfiguration actions (e.g. create a firewall rule, insert an *SwcInfo* object in the list of active components, send a job message to the background worker via a corresponding agent function, ...). The Visitor pattern separates the reconfiguration aspects from the basic functionality of the agent and keeps the agent extensible for new kinds of reconfiguration steps.

For **output processing** and then **input processing** (with a **second *rcHook*** call in

between), the agent goes through its list of software components. For each GPIO-mapped output property of a component the agent reads the contents of a specific file within the container. Then the agent passes the file's contents to the corresponding driver by writing to the corresponding sysfs files. GPIO-mapped inputs are processed in the inverse manner: The agent retrieves the current sensor value from sysfs and writes it into a specific file within the container. The corresponding sysfs files are mirrored to the container in a way which makes it possible to use the MRAA library [Cor18] within containers without access to the real sysfs. Communication-mapped inputs and outputs are not processed individually. Instead, the agent sends a move command message to the barrier qdisc instances via netlink. The barrier module within the network stack then adapts the timestamp associated with the given ingress or egress barrier and releases all buffered network packets with an older timestamp. Message traversal between components is described more in detail later. The input processing phase is followed by a **third reconfiguration hook**. Finally, the agent **triggers the next cycle** by sending an interrupt (the *SIGINT* signal) to all processes within the containers of all active software components. These processes are known from the containers' cgroup files `/sys/fs/cgroup/cpu/lxc/<swc>/cgroup.procs`. To align their cycles with this trigger, the software components have to comply with the behavior described in the following section.

Cyclic Task Execution

Software components are implemented, built, and deployed independently – the deployment unit is an OCI container image together with an LXC configuration file. An arbitrary programming platform can be used for component implementation (C/C++, Python, JavaScript, ...) as long as the resulting container is self-contained and it implements the following behavior. Figure 5.6 illustrates this behavior regarding the cyclic task execution of a ground-state component.

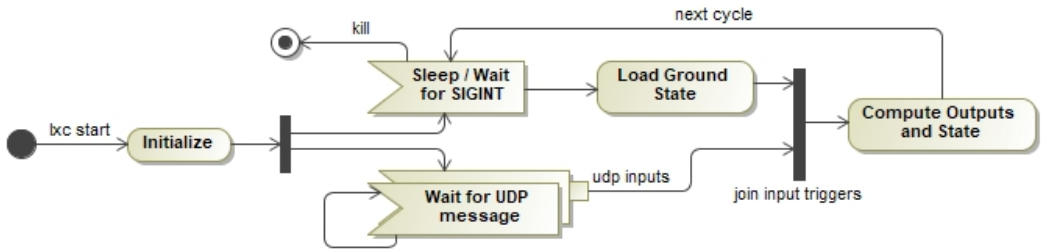

FIGURE 5.6: The cyclic task execution of a ground-state component.

A software component has one task which cyclically computes outputs from inputs. After initializing and before each cycle, the software component waits for a SIGINT signal by the agent (e.g. using *sleep* or a comparable function, depending on the implementation platform). Then the software component reads the inputs from the corresponding files (for I/O-mapped inputs) and from the corresponding UDP messages (for inter-component communication). If the component does have at least one UDP-based input, it can also ignore the signal and instead only wait for the arrival of the inputs. As the agent moves the ingress barrier of the container during input processing before sending the signal, UDP inputs can arrive before the signal, in theory. Most likely, UDP inputs arrive in the container after the signal, with a minimal delay caused by parallel delivery of UDP packets at the beginning of each

cycle. Thus, the software component must join parallel input triggers before beginning the task cycle itself: The signal and UDP packets must be taken in arbitrary order regardless of intra-component multi-threading (e.g. if multiple listeners poll for the different UDP inputs). If the component does not have UDP-based inputs, the agent's signal is the only input trigger. The component implementation should not use time triggers to achieve a periodic execution, except if this is compatible with the signal- and UDP-based task triggering. After receiving all expected input triggers, the component writes the current timestamp to the `_start` file and computes outputs and its new state based on the current state and on its inputs. If it is a ground-state component, the state must be loaded from the component-specific state files in each cycle and only after receiving the signal from the agent. At arbitrary instants during task execution, the component can send UDP outputs and update the contents of the files related to I/O-mapped outputs and the ground state. Due to the container and the egress barrier the outputs are not visible to the outside of the component before the next administration phase. After each completed cycle, the software component updates its ground state (if applicable), writes the timestamp to the `_end` file within its container and waits for the next signal by the agent. An additional heart beat file `_heart` is used during initialization to indicate readiness of the component in two steps. First, after the component is initialized, so the agent can enable real-time priority for the container, and second, when the component successfully requested real-time priority and is ready for the first cycle. As long as the agent does not send a signal or move the ingress barrier, the component must not start another cycle. The agent can stop the container after the last completed cycle using the SIGTERM and SIGKILL signals. In addition to normal cyclic task operation, components can implement the dump/load API: The agent can then alternatively send a dump or load command to the container via a dedicated UDP port if this is supported by the component (i.e., in case of an implicit-state component). In this case, the component dumps or loads its internal state from or to the corresponding state files within its container. This can take multiple cycles without any additional triggering by the agent. During dump and load operations, no task triggers are provided by the agent and the software component should not compute outputs from inputs. Only then the complete quota of the container is used for dump/load and the state does not drift while being processed.

Message Traversal between Software Components

Figure 5.7 shows how UDP is used for inter-component communication in compliance with the logical execution time paradigm. The diagram shows two containerized software components deployed to different nodes. Component SWC1 is the sender of an output message received by component SWC2 as input. SWC1 sends the UDP packet with the computed contents after the agent on node1 triggers cycle n via a signal. The packet is enqueued in the egress barrier installed as the qdisc of the container's virtual ethernet adapter eth0. In the output processing phase of cycle n+1 the agent moves this barrier, so the packet is released from the queue. Through a NAT rule in the egress barrier it is ensured that the packet is transmitted to node2 according to the communication mapping. The egress barrier also ensures that only the configured number of bits per period are released. The packet is transmitted during the execution phase of cycle n+1, redirected from eth1 to ifb0 by a mirred filter and enqueued in the ingress barrier installed as the qdisc of ifb0. The agent on node2 moves this barrier in the input processing phase of cycle n+2, so the packet is released and redirected by another NAT rule at the ingress barrier to the container's

virtual ethernet adapter eth0 via its pair device vethY. The packet is finally received while in parallel the agent triggers SWC2 by sending the SIGINT signal and the UDP packet is processed by SWC2 during cycle n+2. If SWC2 was co-located with SWC1 on node1, then the UDP packet would be enqueued to the ingress barrier on node1 within the same cycle in which it is released – in this example cycle n+1. The steps 5-7 in the activity diagram then happen one cycle earlier, so the packet is processed by SWC2 during the execution phase of cycle n+1. Thus, this implementation is compatible with the behavior modeled in the formal specification in section 3.2.2.

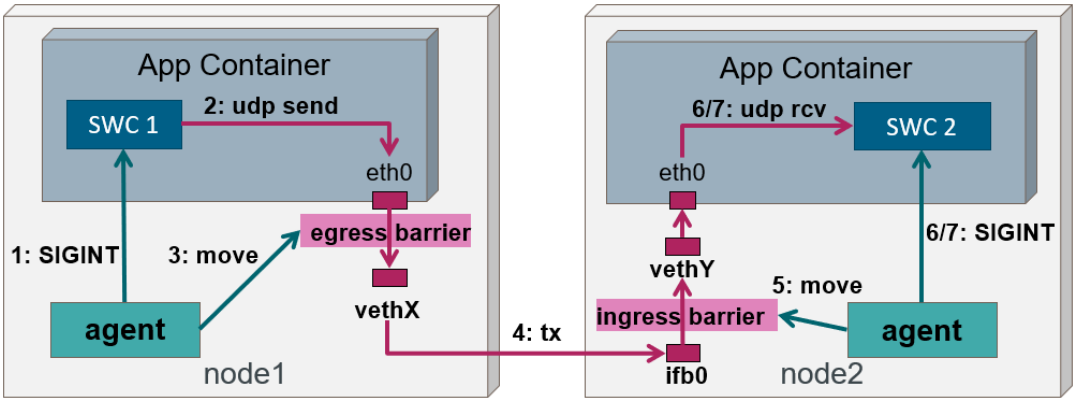

FIGURE 5.7: Schematic activity diagram showing how inter-component communication is conducted step by step.

Start Software Component in Background

We describe the interaction between the agent and the background worker at the example of a *start* step. Three agent threads are involved (see figure 5.8): The agent cyclic routine, the background worker, and the agent listener. The agent cyclic routine initiates the interaction during a reconfiguration hook in the administration phase, if the next reconfiguration step is a *StartStep*. The *apply* function of this step sends a start command containing the container name, its IP address, and the start command to the background worker via a method of the agent. This command message is transmitted to the background worker via the *agentpub* topic using ZeroMQ *ipc*, i.e., via POSIX message queues. While the agent and the software components continue their normal operation, the background worker receives the *StartStep* during the next execution phase. Consequently, the background worker sets up and starts the specified container: It configures tha ARP tables (to avoid ARP requests), installs an egress barrier at the virtual ethernet pair device of the container and then starts the software component using the given command. Then the background worker waits until the software component indicates successful initialization via the heartbeat file. Afterwards, the background worker assigns real-time quota to the container and sends a SIGINT signal to the component. The component requests real-time priority and indicates readiness via the heartbeat file, again. Finally, the background worker sends a done message to the agent listener via the *bgworkerpub* topic using ZeroMQ and waits for the next command message. The done message also contains information about the newly started container such as the index of the newly created eth0 network interface of the container. The agent listener updates the agent's data structure and marks the *StartStep* as done. The agent cyclic routine recognizes the successful completion of this reconfiguration step during the next reconfiguration hook and continues with subsequent reconfiguration steps.

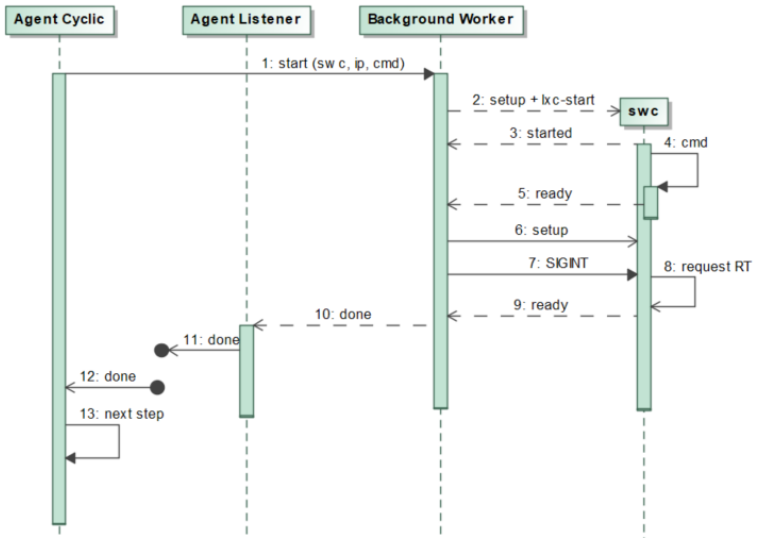

FIGURE 5.8: Sequence diagram of the interaction between the agent's cyclic routine, the background worker and the agent listener to start a container in background.

System Start

The distributed control system can be started by executing the real-time container agent binary on the master node via SSH. Prior to this, all nodes have to be configured and booted accordingly and the clocks must already be synchronized via PTP to below 1 ms master offset. The agent first loads the agent configuration file `/etc/rtca/agent-config.yml`. The first configuration item is the log level, which the agent sets as configured. Then it detaches from the console using the *daemonize* function (to reduce network traffic). Then the agent sets up its ZeroMQ topics for communication with the background worker. The agent also initializes the I/Os as needed. Afterwards, the agent parses the reconfiguration plan from the file refered to by the configuration. If the agent is configured as master, it now starts the real-time container agent binary on the slave nodes via SSH. Then the agent sets up the system environment. In this step the agent loads the kernel module for the barrier qdisc, initializes the netlink communication with the kernel's network stack and ensures that the custom `RTCA_BARRIER` family is supported (i.e., that the agent runs on a supported system).

The agent starts the agent listener in a new POSIX thread, which opens an admin communication port and starts polling incoming intra- and inter-agent messages (UDP packets and ZeroMQ messages). The agent forks the background worker, which moves to a dedicated cgroup, sets up its ZeroMQ topics, and then starts processing incoming command messages. If the agent (back in the main process/thread) is configured as master, it waits so that the agents on the slave nodes are passed this point of initialization. The agent starts another POSIX thread for the cyclic routine. To setup isochronuous cycles across nodes, this process first waits for the next full second of the system clock using *gettimeofday* and *usleep*. Then it switches the scheduling policy to `SCHED_DEADLINE` with the configured parameters *period*, *runtime* and *deadline*. Afterwards, the agents synchronize across nodes to identify a global first cycle: The master's cyclic routine sends a start time message to all slaves. All cyclic routines wait for the first cycle in which the start time is reached and then synchronously switch to the system execution mode. To make this work,

the agents' periods must be set to a fraction or multiple of a second, as this is currently the only supported time grid. In the system execution phase, the agent's cyclic routine performs the administration phase in each cycle as specified beforehand. In the first cycle, no software component is running, yet. The distributed embedded application now needs to be initialized according to the previously parsed reconfiguration plan. In general, the initialization plan should start all components, then synchronize across nodes and finally create the mappings and activate the components (see the blueprint described in section 4.3.4). While the created processes and threads continue to operate the distributed embedded application, the agent's main thread waits for triggers to stop the system. Currently, a fixed system execution duration can be configured. Alternatively, the main thread stops the system in case a termination request is issued by the DevOperator (effectively, a SIGTERM signal).

5.2.4 Deployment View

We describe the technical infrastructure used to execute the distributed control system and the mapping of building blocks to that infrastructure elements. This view only shows the decomposition on a single node, which are relevant for the multi-node aspects. We do not specify a concrete deployment, but rather a deployment and topology scheme, which must be instantiated for application use cases.

Network Topology

Figure 5.9 shows a topology with two MCU nodes and an engineering system node. In general, the network topologies should consist of one engineering system and at least one MCU. On the engineering node, all complementing engineering and monitoring subsystems are deployed (see section 5.3), most of them running in Docker containers (e.g. container registry, development container, log collector, SSH client, dashboard server, update management server, ...). The distributed embedded application is running on the MCU nodes, only. We use SIMATIC IOT2040 MCUs, which have an x86-based System-on-a-Chip (SoC) with 400 MHz and 1 GB RAM. The root file system is provided by industrial-grade SDHC microSD cards with 32 GB. Besides diverse other interfaces, the IOT2040 has two ethernet adapters with 10 MBit/s: *eth0* and *eth1*. Each MCU node is reachable from each other (directly or indirectly) via *eth1*, the plant network for inter-agent and inter-component communication. The plant network is important for our platform concept: Components must not be able to define communication relationships without communication mappings from outside their containers and we must be able to bandwidth-control the messages and reconfigure the routing/orchestration. The following network scheme helps to make this possible in our evaluation platform.

The plant network has the network ID *192.168.0.0/16* and the network interface *eth1* is assigned the IP address *192.168.n.1*, where *n* is the numeric identifier of the *MCU* node (e.g. *node1* has *192.168.1.1*). Within each node, the LXC-based containers are assigned IP addresses starting from *192.168.n.100* upwards (e.g. the first container on *node1* gets *192.168.1.100*, the second one gets *192.168.1.101*, ...). These IP addresses are assigned to the container's virtual ethernet adapter *eth0* (which is in another network namespace than the host's *eth0*). Outside the containers, the outer interfaces of the veth pair device are enslaved to a virtual network bridge *lxcbr0* on each node, which have the IP addresses *192.168.n.2*. The routes are configured so that all outgoing traffic from within a container and all incoming packets to local

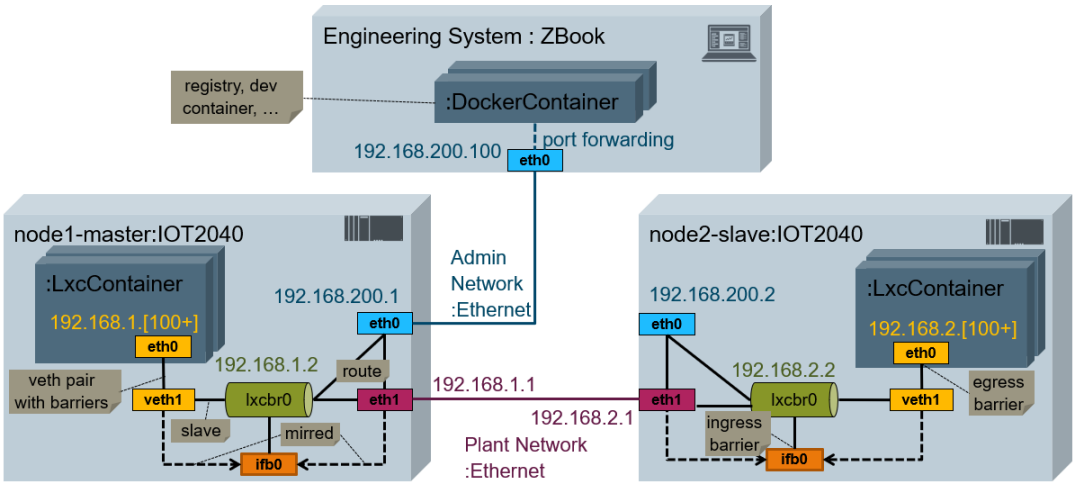

FIGURE 5.9: The network topology used if not stated otherwise: Two SIMATIC IOT2040 MCU nodes are connected via ethernet over their *eth1* interfaces. The containers on each node are reachable via a virtual network bridge *lxcbr0*. The master MCU *node1* is connected with the engineering system over its *eth0* interface and acts as gateway for administration communication with *node2*.

containers are sent via this local bridge as gateway. Such incoming packets are redirected from eth1 and the outer veth devices to ifb0 based on their destination IP address *192.168.n.2*. On other nodes and outside of the containers, all traffic with destination subnet *192.168.n.0/24* is sent via gateway *192.168.n.1*. Note that our networking concept is complemented by the barrier qdiscs installed at the veth pair devices of the containers and at ifb0: These qdiscs additionally apply NAT rules, which modify the destination of the packets according to the communication mapping, and shape the traffic accordingly (logical execution time and bandwidth control). Thus, a packet's destination is modified by the egress qdisc to the address of the target node's *lxcbr0*, and by the ingress qdisc to the address of the target container's *eth0*.

Apart from the plant network, we use a separated administration network for communication between the engineering system and the *MCU* nodes. During engineering and configuration time and at least for starting the distributed control system, the engineering system must be connected to at least one MCU node, e.g. the master node. For monitoring the system and for triggering reconfigurations during operation, the connection is needed as well. The administration network has the ID *192.168.200.0/24* and the *MCU* nodes are assigned the IP addresses *192.168.200.n*, with the nodes' numeric identifiers *n*. The engineering system has the IP address *192.168.200.100*. Depending on the exact network topology of the use case, each *MCU* node can be connected directly or indirectly – either via a switched administration network or using SSH forwarding over the plant network. During system operation the SSH bandwidth of the plant network must be throttled. Thus, the variant with a completely separated, switched administration network to all nodes is faster. However, the indirect approach does not only require less hardware, but it also enables administration via a single direct connection to one of the MCU nodes, only. When not stated otherwise, we use the indirect approach.

Plant Network Communication

Figure 5.10 shows the required plant network communication from the different layers of the distributed control system. On system level, the PTP messages must

be exchanged between the MCU nodes at highest priority, while SSH-based communication (starting slaves, downloading container images, ...) should be strictly bandwidth-constrained (planned). On the platform-level, UDP-based inter-agent communication for the coordination steps and start-up synchronization is needed rarely and on low volume but at high priority. Finally, the software components of the application layer must communicate periodically with application-specific volume and paths within the plant network. This is implemented by following traffic control concept: All application-specific communication is bandwidth-controlled by the barrier qdiscs. The inter-agent communication related to coordination steps and PTP are sent via unrestricted qdiscs. All other communication is enqueued to Hierarchical Token Bucket (HTB) qdiscs, which throttle the throughput down to a configurable quota (planned). The background worker is throttled down by a dedicated qdisc (planned) so that potentially ongoing monitoring communication does not take away quota needed for time-critical state transfer operations. In this way we can ensure that there is enough quota for the inter-component and inter-agent communication. Note that this prototype does not use time slices, yet, which leads to a higher probability that UDP messages are lost.

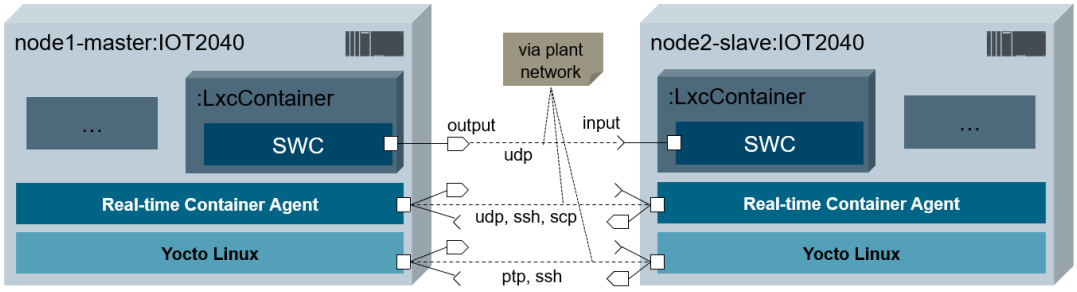

FIGURE 5.10: Remote communication over the plant network between two MCU nodes: The Ethernet connection is basically shared by PTP and SSH in the system layer, inter-agent communication in the platform layer based on UDP, SSH and SCP, and UDP-based inter-component communication in the application layer.

5.2.5 Cross-cutting Concepts

We describe noticable additional aspects of the evaluation prototype, which cut across the previous views and affect multiple parts of the system. To understand the concept realization and evaluation results following aspects are most relevant: How components, distributed control systems and reconfigurations are modeled (in terms of description and configuration) and how the different reconfiguration steps are implemented. While configuration might seem to be an unimportant side topic, note that this is how the evaluation platform can be adapted to run a specific distributed control system. This aspect is crucial for our concepts, because – while all the technical means are needed as basis and enablers – the platform configuration is where we can finally leverage simplified integration, reuse, and reconfigurability. Conceptually, the component description and configuration files are the link between the design-time model and the runtime model. They provide an extract of the design-time model information, so that the nodes and components can be configured and operated accordingly at runtime. In the current prototype, these files are not yet generated from a higher-level design-time model on the engineering system. Instead, the DevOperator must create them accordingly on the corresponding nodes and even run some setup commands manually.

Container Configuration

For each software component instance, an appropriate LXC configuration file is needed on the corresponding node to configure the application container. Listing 5.3 shows a partial example for the *ButtonController* on *node1* in the *onBtnSwitch* example system. It refers to the rootfs folder (usually next to the configuration file), which contains the executables and all auxilliary files of a component. The quota per fixed period are configured as required by the component (e.g. by its *wcet*) – however, this needs to be done by the background worker to control the instant at which real-time priority is available (and these settings are not supported at least by the used LXC version). Via the LXC configuration, a virtual ethernet pair device is partially configured as described before in section 5.2.4. The remaining network configuration aspects are done programmatically by the agent at reconfiguration time according to the information within the reconfiguration plan (especially static ARP configuration and barrier qdisc configuration). Note that the container configuration should be done automatically by the agent at reconfiguration time, too. It is done manually (which is laborious and error prone) only because this has not been automated, yet. When automating this in future, at least following two aspects must be considered: First, while most of the configuration entries can be seen as standard boilerplate, there are some differences depending on the programming platform (e.g. python), so platform-specific templates are needed. Second, the configuration generation must be done in background.

```
# /var/lib/lxc/button-v1/config
lxc.rootfs = /var/lib/lxc/button-v1/rootfs
lxc.rootfs.backend = dir
lxc.haltsignal = SIGUSR1
lxc.rebootsignal = SIGTERM
lxc.utsname = button

#lxc.prlimit.rtprio = 42                  # set by bg worker
#lxc.cgroup.cpu.cpu.rt_period_us = 50000  # set by bg worker
#lxc.cgroup.cpu.cpu.rt_runtime_us = 10000 # set by bg worker

lxc.network.name = eth0
lxc.network.type = veth
lxc.network.link = lxcbr0
lxc.network.hwaddr = 00:16:3e:3a:01:01
lxc.network.ipv4 = 192.168.1.100/24
lxc.network.ipv4.gateway = 192.168.1.2
# ...
```

LISTING 5.3: Partial LXC configuration file for the *ButtonController*'s container.

Agent Configuration

Listing 5.4 shows an agent configuration file for adjusting the agent's execution. Using the **logLevel* entries the agent's and the barrier's log levels can be set according to the syslog standard [Ger09] (e.g. 7 or `LOG_DEBUG` corresponds to debug-level messages). A termination trigger can be configured using the *agentRuntimeSeconds* entry: In the given example the agent terminates system execution after 90 seconds. The scheduling of the agent's cyclic routine can be configured via three entries, which correspond to the parameters of the `SCHED_DEADLINE` scheduling policy: *period_ms* sets the duration of a cycle, *runtime_ms* sets the quota available for this routine per period, and *deadline_ms* sets a deadline within the period. In this case, the agent is configured for a 100 ms cycle with 10 ms reserved for the administration phase. Though we want the deadline to be 10 ms after each isochronuous period begin according to the runtime, we still set it to the full period to avoid any shifting of the isochronuous ticks by the "replenishment" behavior of the scheduler. These

configurations should be set equally for all agents across nodes. Besides other, minor configuration options not shown in this listing, the final relevant entry is *nodeConfigFilePath*, which holds configuration options for a specific node in a specific network topology.

```
# agent-config.yml
logLevel: 7
barrierLogLevel: 4
agentRuntimeSeconds: 90
sched_deadline:
    period_ms: 100
    runtime_ms: 10
    deadline_ms: 100
nodeConfigFilePath: "node.yml"
```

LISTING 5.4: Essential parts of an agent configuration file.

Listing 5.5 and listing 5.6 show the node configuration files for the two nodes in the *onBtnSwitch* example system. The *nodeId* entry sets the numeric identifier of the node and implicitly its hostname. The agent on *node1* is configured as *master* with *node2* in its list of slaves and the agent on *node2* is configured as *slave* with *node1* as its master. Consequently, the agent on *node1* will start the agent on *node2* and determine the system start time. For configuring the network, the agent needs the information about the node's plant network IP adress and the IP adress of the virtual ethernet bridge for the containers. Additional settings including agent-managed GPIOs can be configured. Finally, a reconfiguration plan can be referred to via the *initialRcPlanFile*. The agent loads the reconfiguration plan before system execution and starts applying the steps in the first cycle. Only via such a reconfiguration plan, the agent can be instructed to automatically launch a specific set of components at system start time. If no initialization plan is specified, the agent idles until it stops due to a termination trigger or until a dynamic reconfiguration plan is provided by the engineering system. Thus, the agents are configured for a specific deployment of a distributed embedded application via reconfiguration plans (see next section).

```
# node1.yml
nodeId: 1
role: "master"
slaves: ["node2"]
eth1:
  ip: 192.168.1.1
lxcbr0:
  ip: 192.168.1.2
# ...
initialRcPlanFile: "start-obs.node1.rcplan"
```

LISTING 5.5: A master node configuration file for *node1*.

```
# node2.yml
nodeId: 2
role: "slave"
master: "node1"
eth1:
  ip: 192.168.2.1
lxcbr0:
  ip: 192.168.2.2
# ...
initialRcPlanFile: "start-obs.node2.rcplan"
```

LISTING 5.6: A slave configuration file for *node2*.

Reconfiguration Plan Description Language (RcPlan DSL)

The agent can parse reconfiguration plans from an RcPlan DSL input using an RcPlan Parser component. In essence each line in an RcPlan DSL file defines a reconfiguration step. Listing 5.7 and listing 5.8 show initialization reconfiguration plans for the *onBtnSwitch* example system. They are referenced by the node configuration files described before. The plans correspond to the reconfiguration blueprint for starting a distributed embedded application (see section 4.3.4): After the application containers have been started, the two nodes perform a two-way handshake. Then they synchronously start triggering the new components and processing their inputs and

```
# start-obs.node1.rcplan
start "button-v1" "192.168.1.101" "button-client 2000" quota 10000
syncPoint 1 wait "node2"
syncPoint 2 notify "node2"
offset 0:
    betweenIO:
        addSwc "button-v1"
        addGPIO "button-v1" "btnIn" "in" "63"
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
```

LISTING 5.7: A reconfiguration plan to start the *ButtonController* component on *node1* installed as container *button-v1*.

```
# start-obs.node2.rcplan
start "blink-v1" "192.168.2.100" "blink-server 2000" quota 10000
syncPoint 1 notify "node1"
syncPoint 2 wait "node1"
offset 2:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
        addSwc "blink-v1"
        addGPIO "blink-v1" "ledOut" "out" "7"
```

LISTING 5.8: A reconfiguration plan to start the *LedController* component on *node2* installed as container *blink-v1*.

outputs along the dataflow of the distributed embedded application. The examples show that the step definitions in the RcPlan DSL are similar to the constructor functions in the formal specification in section 4.1.

RcPlan DSL files can be provided via the *initialRcPlanFile* configuration item at system start time or dynamically from the engineering system. If the plan is provided at system start time, it is parsed during agent initialization prior to the first cycle. Otherwise, this is done in background by the background worker on command by the agent cyclic routine like a reconfiguration step. The agent sends the command if there is a new reconfiguration plan and the current reconfiguration plan is done (see listing 5.2). After parsing, the background worker sends the parsed reconfiguration step objects to the agent listener one by one to minimize interference with the embedded application. The agent listener appends the steps to a temporary list, which becomes the reconfiguration plan after all steps have been appended. The swap is done in the agent's cyclic routine, so, no locking is needed for this procedure.

Reconfiguration Step Implementation

Table 5.7 describes how the reconfiguration steps are implemented in the evaluation platform. Most steps directly comply with the formal specification in section 4.1. For evaluation purposes, two additional steps *loop* and *endLoop* were added, so that we can repeat reconfiguration plans (e.g. upgrade and downgrade multiple times) to gather runtime statistics.

Step	Implementation
loop	Clears the *rcLoopPlan* list of reconfiguration steps and sets the *loopCount*. While the *loopCount* is non-zero, all completed steps until the *end* step are pushed to the *rcLoopPlan* list, except steps with the *once* flag set (*endLoop* and *waitForCycle*).

Step	Implementation
endLoop	Decreases the *rcLoopCount* and if it is greater than zero it resets the steps in the *rcLoopPlan* list (e.g. mark them as *todo*) and pushes them back to the *rcPlan* list, effectively starting a new iteration.
notify	Calculates a continuation cycle number *cc*, sends it to the agent listener on the specified node via UDP and inserts an absolute *wait* step.
wait (notify)	Looks up a matching synchronization message in the *syncPoints* list of the agent. The message is pushed to this list by the agent listener after reception. If the message is found, it is removed, an absolute *wait* step is inserted, and the step is marked as done.
wait (abs)	Marks itself as done as soon as the agent's *cycle* counter matches.
wait (rel)	Replaces itself with an absolute *wait* step based on the agent's *cycle*.
download (planned)	Sends a download command message (containing the image name and the container name) to the background worker. The background worker pulls the image from the container registry (not implemented, e.g. tarball via wget, podman pull ...) and sends a done message. The agent listener marks the step as done.
start	Sends a start command message (containing the container name, the IP address the command to execute, and the quota) to the background worker. The background worker sets up the required tmpfs-based mount folders (e.g. the fake sysfs), starts the container via *lxc-start*, waits until the container state is *RUNNING* and the heartbeat file exists. Then it sets up the network namespace, statically configures link-layer routing via *lxc-attach* and *arp*, and installs the egress barrier qdisc at the inner veth device via *tc*. The worker assigns real-time quota via the container's cgroup files and signals SIGINT to the container's processes. When the heartbeat file is recreated by the component, the worker finally sends a done message (containing the indices of the veth pair device). The agent listener updates the component state and marks the step as done.
stop	Sends a stop command message (containing the container name) to the background worker. The background worker destroys the egress barrier instance, cleans up the namespace, and stops the container via *lxc-stop*. After the container state changes to *STOPPED* the worker sends a done message. The agent listener updates the component state and marks the step as done.
destroy (planned)	Sends a destroy command message (containing the container name) to the background worker. The worker destroys the container via *lxc-destroy* and sends a done message. The agent listener clears the component record and marks the step as done.
addSwc	Marks the component in the *swcs* list as *RUNNING* so it is triggered.
rmSwc	Marks the component in *swcs* list as *BLOCKED* so it is not triggered.
addCom	Creates a NAT rule at the right barrier via *libnl* and adds the property to the component in the *swcs* list so the barrier is moved. The agent can also set a logical addTime if needed (see section 4.3).
rmCom	Deletes the NAT rule and the property from the barrier and *swcs* list. If needed, the rule is only marked as deleted by setting the logical rmTime (see section 4.3).
addMsg	Stores the message info in the *msgs* list (used by addCom and rmCom).
rmMsg	Deletes the message info.

Step	Implementation
addIO	Adds the I/O mapping information to the component entry in *swcs* so it is processed. The agent must be pre-configured for the I/O.
rmIO	Removes the I/O mapping information from the component entry.
extract	Copies the file content from a container to the host using C++ *iostream*. If *zet* flag is set, this is done directly within the reconfiguration hook. Otherwise, the agent moves the step to the *bgStep* list (so it does not block further steps) and sends an extract command message (containing the container name and paths) to the background worker, who copies the file and sends a done message.
transfer	Copies the file content from one container to another one like *extract*.
inject	Copies the file content from the host to a container like *extract*.
transmit	Moves the step to the *bgStep* list and sends a transmit command message (containing the container name and path, node and remote path) to the background worker. The worker sends the file info and contents to the agent port on the given node via UDP and sends back a done message. The agent listener on the remote node creates the file on reception. A redesign is planned to reuse existing solutions, but RCP and SCP took too long for sending a small state within a single period.
dump	Moves the step to the *bgStep* list and sends a dump command message (containing a path within the container) to the component. The component creates a dump file from its internal state without additional triggers using the container's quota (the task must be blocked during this time) and sends back a done message.
load	Like *dump*, but instead of dumping to the file, the component updates its internal state from the given state file.
capture	Depends on the mapping: For an I/O-mapped property, the I/O file content is copied from the container to a file in the host. For a communication-mapped property, a capture command is sent to the barrier (containing at least the container name, the source/destination port number, and a capture name). The barrier creates a capture entry and will store clones of the matching network packets at dequeue time. To stop capturing, the agent sends a stopCapture command at the start of the next period. The agent sets a drop flag for inputs if the component is blocked, so matching packets are captured and dropped.
replay	Reverse operation of *capture*: For I/O-mapped properties, this copies the contents from the referenced file to the I/O file within the container. For communication-mapped properties, this sends a replay command to the barrier, which enqueues clones of the captured packets updating their timestamps.

TABLE 5.7: How the reconfiguration steps are implemented in the prototype.

Logging and Tracing

We describe how the prototype addresses the functional requirements M-6 (logging of platform and eApp execution), M-7 (measure and report real-time behavior), and O-2 (measure resource consumption). These features are essential for development, demonstration and evaluation of the platform. However, as they influence the system's performance and temporal behavior, logging and tracing must be tailored to

the needs of a run.

The platform and application components provide human-readable log messages for debugging and parsable log information for the system monitoring dashboard. An optional rsyslog daemon is running on each node. The agent sends log messages to this daemon using *syslog.h*, i.e., via the standard unix domain socket */dev/log*. As described before, the log level is configurable at system start time via the agent configuration file. The agent sets its own log level programmatically via the *setlogmask* function. The barrier uses printk to log to the kernel log. The agent sends the configured barrier log level to the barrier via netlink. The containerized components log to *stdout*, which is redirected to a logfile configured by the agent via the `-L` option of the *lxc-start* command. The log levels of an eApp or individual components are not (yet) configurable via the agent configuration file, but the component binaries support a `--silent` flag, which can be added to the start command in the reconfiguration plan. The DevOperator can use the *rsyslog* configuration file to define how log messages (from socket and/or file input) should be handled. Depending on the configuration, log messages may be collected locally under */etc/rtca/* and remotely on the engineering system. Log forwarding is needed for the system monitoring dashboard and for the update management system. Table 5.8 shows the logging setups used for different run types.

setup	**log forwarding**	**log levels**
dev	as needed	*DEBUG*
demo	everything	*INFO*
eval	disabled	*ERROR*

TABLE 5.8: Overview of the different logging setups.

For measuring the real-time behavior of the system, we use an additional stopwatch component within the agent. The stopwatch measures the wall times needed for specific code sections. It has a fixed reserved memory (i.e., numbers of measurement series), which is allocated before the benchmarking. For one measurement, the start and stop methods of the stopwatch must be called with the measurement bucket identifier. The stopwatch then gathers two timestamps using *gettimeofday* and stores them for later. Alternatively, the timestamps can be given by the caller, which the agent does on behalf of the software components during the health check. The statistics (count and duration min/max/sum_squared) are updated continuously from the individual measurements and reported as part of the termination procedure of the agent. This way they are automatically persisted locally and optionally forwarded to the engineering system (logging of individual measurements is too costly, though). If extended statistics are requested, the stopwatch also maintains statistics for the drift from given start times and slack time to given deadlines. This way the stopwatch measures the execution times of the administration phase steps (health check, rcHook beforeOutputs, input processing, ...), of the different reconfiguration step kinds (addCom, rmCom, addMsg, rmMsg, ...), and of the software components' tasks. We plan to extend the agent with resource monitoring features to get a high resolution at lowest impact. For instance, we want to use the stopwatch to measure the CPU time used for the different buckets, too. As long as this is not fully implemented, tracing of resources can only be done with general tools built into our custom Linux image. As an example: We use *top* [War20] to measure CPU time and memory usage of the different processes. However, the resolution of *top* is 100 ms and there is no possibility to align it with the cycle.

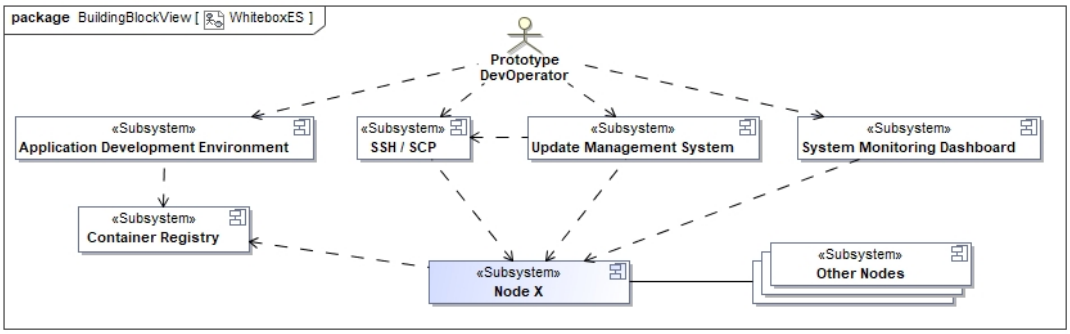

FIGURE 5.11: Overview of the engineering system components used to develop and operate distributed control systems using our prototypical runtime platform.

5.3 Complementing Systems

We describe the engineering system components complementing the runtime platform in the real-time container architecture (see figure 5.11). Engineering system components are also important for the proper execution of distributed control systems (besides the runtime platform itself): During development and engineering, we have to ensure that model and implementation are correct and feasible (as described in sections 3.3 and 4.2), while leveraging simplified integration. At operation time, the engineering system components offer visibility and control over the distributed control system and provide reconfiguration plans and container images to the nodes. While these aspects are of highest importance for an industrial-grade platform, this prototype only provides a minimum viable engineering system: Instead of integrating with common engineering and SCADA systems, we prefer more pragmatic solutions to be able to conduct the evaluation and to demonstrate the platform behavior.

5.3.1 Application Development Environment

The application development environment is a subsystem on the engineering system used to implement, build and package software components and to create distributed embedded applications from them. It is thus also the place for debugging, testing, simulation and verification. Our main prototyping efforts were located in the runtime platform layer and not in the application layer. Thus, the current prototype does not provide sophisticated solutions for all of the mentioned engineering tasks. However, this description gives an understanding of our work and indicates the potential of our platform architecture while pointing out what is really covered in the prototypical realization.

Code Development: For developing both software components and platform components we used Visual Studio Code with the Remote Containers extension and language-specific development extensions (C/C++, Python). With these extensions and an appropriate development container, software components can be implemented on various programming platforms. For example, for C/C++ components we used an Ubuntu-based Docker container with our yocto-based cross-build SDK installed (GCC, GDB . . .).

Testing and Debugging: For black-box testing and debugging of software components and eApps they must be executed on the real target system. To make this possible within a development container we will have to solve the simulation of I/Os, the dependence of the runtime platform on the barrier kernel module or porting of

the barrier, and real-time scheduling within the development container. Lacking a virtual runtime system it is difficult to debug the code by stepping through it: Even if it was possible to pause the platform and all software components on a node consistently with the real-time scheduling – in the real environment this disrupts the system's timing and thus its embedded functionality. Instead, we often used the debug log messages created on the target system to locate bugs in the code. Of course, unit tests can be done on the engineering system with language-specific means within the scope of a component to test individual functions or classes. Usually, components were developed without the runtime platform at first and then containerized and used in a single-node test deployment to test the interplay with the platform and equipment after integration with other components.

Simulation: The DevOperator can simulate the reactivity and the functional behavior of a distributed control system using Maude as described in section 3.3. For this task we also used Visual Studio Code and the Remote Containers extension together with the Maude extension (for basic syntax highlighting) and a Docker container with the Maude executables and our Maude model installed. To simulate concrete systems and reconfigurations, the models of the concrete application, the reconfiguration plan and the environment of the system are needed in addition to our runtime model. The formal simulation does not fully replace execution of the real code with simulated I/Os. However, the real-time system can be simulated accurately under the assumption that the temporal bounds and component implementations were modeled correctly.

Verification: Using the Maude model and the proposed dataflow graphs we can formally analyze the correctness, feasibility, reactivity, and functional behavior of a system as described in sections 3.3 and 4.2. Note that most of these analyses target the structure of the model, only, because the state space of our model is too large to run an LTL or TCTL model check in Maude in practice. The dataflow graph construction and analysis must be done manually as of today – it is not automatically constructed from the Maude model or configuration files.

Packaging: After successfully building a software component it needs to be packed as a container image compatible with LXC. For this purpose, we use Docker: A component-specific Dockerfile describes the container image (i.e., installation of dependencies, cross-building with our Yocto SDK, adding configuration files, ...). Building and exporting it results in an OCI-compatible container image supported by LXC on the target. Alternatively, simple components were also cross-built and uploaded to the rootfs of a new container on the target created with *lxc-create* and the busybox template, for example. In all variants, the corresponding LXC configuration file is needed in addition. At the moment, it is not generated and thus needs to be provided in addition. It must be consistent with the deployment (e.g. the IP address), which should be solved by a generative approach in future.

5.3.2 System Monitoring Dashboard

For testing and demonstration purposes, we developed a system monitoring dashboard based on the Elastic stack (Elasticsearch, Logstash, Kibana). Figure 5.12 shows the basic architecture of this engineering system component. The rsyslog service running on each MCU node forwards specific log messages to the engineering system. Then the log messages are processed by a Logstash instance, which detects key-value-pairs and adds them to the JSON objects representing the messages. These

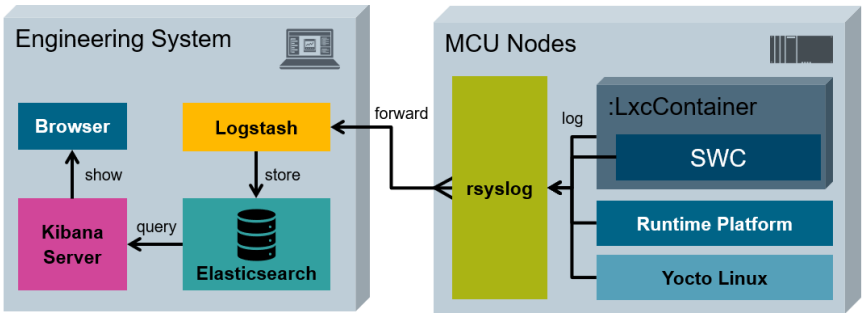

FIGURE 5.12: High-level architecture of the system monitoring dashboard and the log dataflow from the embedded system via the rsyslog service through the Elastic stack components to the browser-based dashboard application.

objects are then stored in the Elasticsearch document database. Finally, Kibana is used via a web browser to explore the logs using custom queries and to create and view visualizations for them. Following visualizations were defined to visualize the temporal behavior during a run of the prototype:

- The wall times (y-axis) needed for the different agent steps per cycle (x-axis) as stacked bar charts (see figure 5.13).
- The wall times of the reconfiguration steps per cycle (x-axis) as stacked bar charts. This also shows the absolute cycle numbers in which reconfigurations were performed.
- The drift of the real cycle start times from the scheduled cycle start times in milliseconds (y-axis) for each cycle (x-axis). This is primarily a platform development metric and should always be close to zero if no error occurs.

These metrics show the temporal behavior of the runtime platform, but not the application behavior. Application-specific visualizations could be added to the dashboards as needed, e.g. to show the inputs and outputs of the system over time. To gather such data, the applications must create log messages in the key-value format and their log messages must be configured for forwarding. Especially in demonstration use cases, such metrics can show that (or whether) the distributed control system continuously works during dynamic reconfigurations. Obviously, 3D simulations of the technical process controlled by the distributed embedded application would greatly improve such demonstrations. However, these visualizations were only conceptual or prototypical at most and will need to be worked on in future. For development purposes, we used different command line monitoring tools via SSH depending on the area of interest. The information from these tools should be added to future monitoring extensions, too, for example:

- Monitor container states: `watch -n1 lxc-ls -f`
- Monitor full agent log, including component health and reconfiguration progress: `tail -f /etc/rtca/agent.log`
- Monitor barrier via kernel log: `dmesg -w`

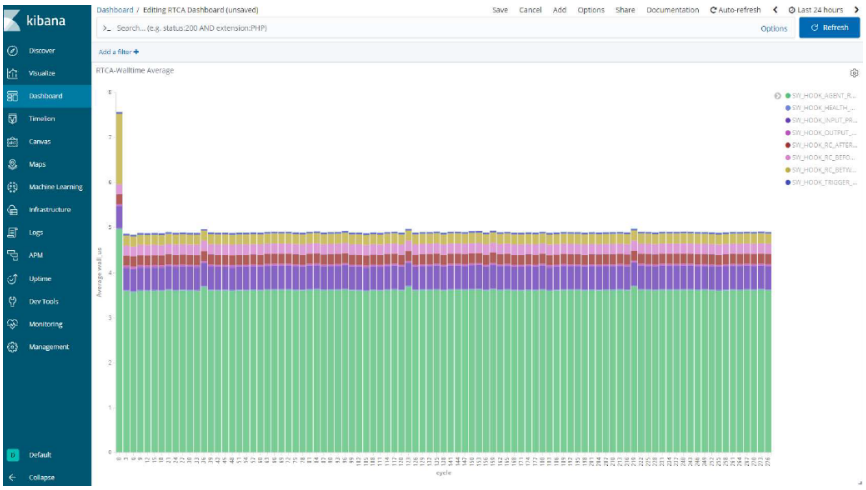

FIGURE 5.13: Screenshot of the system monitoring dashboard.

5.3.3 Update Management System

Today, launching a distributed control system and triggering reconfigurations is only possible via command line. The agent is started on the master node via SSH and pulls up the distributed control system according to the configuration. During operation, reconfigurations can be triggered by sending reconfiguration plans to the agents via a ZeroMQ interface, for which we have a small Node.js based client tool. To make this more operator-friendly and less error-prone, we are working on an update management system. We briefly outline our ideas to show how reconfiguration complexity could be hidden from the DevOperator. According to the current design, the update management system is basically a wrapper around the SSH solution. First, it will need to show monitoring information such as whether it is running, the reconfiguration progress, and the state of the software components on the different nodes. The states could be queried from the Elasticsearch component of the system monitoring dashboard (see section 5.3.2). On top of that, the embedded system and independent deployment units need to be started, stopped and updated via corresponding UI widgets. In an event-based system, newly started software components would dynamically find their communication partners via pub/sub, so Kubernetes could be used, for instance. In the RTCA prototype, however, the dynamic wiring of the containers is more complicated (but also: under control and time deterministic). Thus, the UI features for starting, stopping, and updating must somehow lead to corresponding reconfiguration plans being pushed to the MCU nodes instead of simple commands. For some reconfiguration scenarios covered by a blueprint (see chapter 4.3), those plans could be generated automatically, e.g. for updating individual components in a backwards-compatible way. However, we can not derive consistent and reactive reconfiguration procedures for all update scenarios, yet (consider state transfer leading to quality degradation). For more complex changes to a running control system, manually designed reconfiguration plans will be required. Section 4.2 elaborated on how such plans can be analysed regarding consistency and reactivity. Features for this analysis should be integrated in the update management system, too, because the DevOperator will sometimes need to make a trade-off between consistency, reactivity, and evolvability.

Chapter 6

Evaluation of the Platform Concepts and Prototype

Figure 6.1 shows how we evaluated our modeling and runtime platform concepts using our Maude model (see chapters 3 and 4) and our platform prototype (see chapter 5). On top of the platform we modeled and implemented the *onBtnSwitch* example system and the *CubeBot* system. The *onBtnSwitch* system is a minimum viable distributed control system first described in [TSK18]. Its simplicity minimizes the application-specific efforts needed to model and implement it. We used it during platform specification and implementation, as a running example, and for platform evaluation. For this evaluation we used the *onBtnSwitch* system in different variants and reconfiguration scenarios to gather runtime statistics about the platform. These measurements indicate whether the platform concepts can be used in presence of given requirements (e.g. period, reaction time ...). Additionally, we performed a complete round trip from model to running code for the *onBtnSwitch* system – including reconfigurations. In this complete case study we can compare the simulated system behavior with the real system behavior to identify whether the model is useful to describe and analyze the real system.

The *CubeBot* system [HLL+22] is a smart factory showcase system, which has more complexity within the application and uses more complex equipment. This adresses the question whether our engineering and runtime concepts are suitable for a realistic distributed embedded application. Primarily, we were interested to see and show whether and to which extent more realistic technolgies expected in modern automation can be used in compliance with our runtime concepts. Additionally, we evaluated whether the different engineering steps were decoupled as proposed in chapter 3. The case study includes a reconfiguration scenario to also evaluate the feasibility of our reconfiguration concepts for a realistic system. Thus, the two case studies complement each other: Gathering runtime statistics and comparing the real system with the simulation was done with the *onBtnSwitch* system. The suitability of our engineering and runtime concepts for real distributed control systems was evaluated with the more challenging *CubeBot* case study.

6.1 Case Study: *onBtnSwitch*

The *onBtnSwitch* system was already introduced in section 1.2, as it was used as running example throughout this thesis. In essence, when a button on the first node is pressed (rising flank), then the state of an LED on the second node should be inverted (on/off). Many fragments of the corresponding model were used to illustrate the formal specification in the chapters 3 and 4. Thus, we only give a compact overview of the model and implementation for the different variants of the

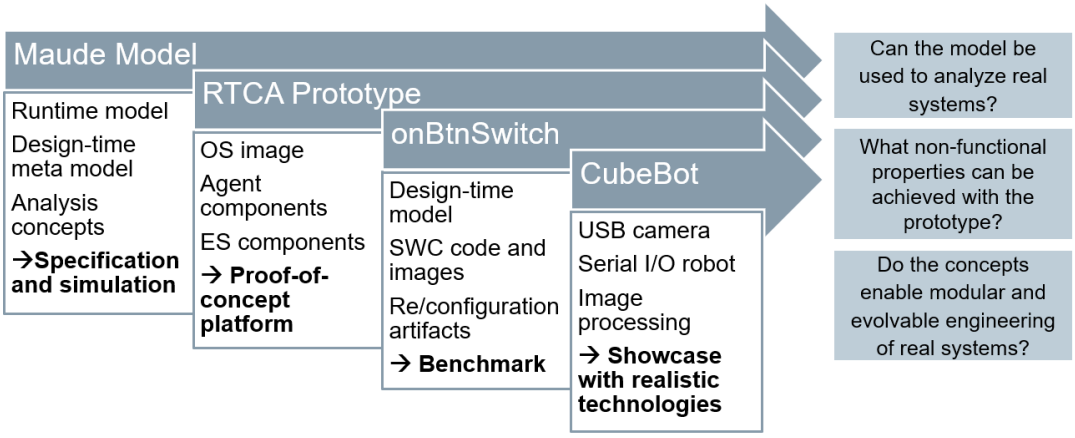

FIGURE 6.1: Schematic overview of the evaluation.

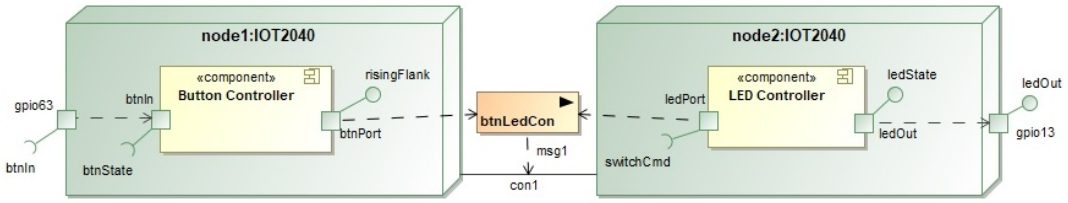

FIGURE 6.2: Diagram of the abstract deployment for *onBtnSwitch* (cf. Section 3.1.4).

onBtnSwitch system and of its reconfiguration. Then we compare the model-based simulation and the real execution of the prototype for the different variants. Afterwards we describe and evaluate the non-funtional properties of the prototype, which we measured in a benchmarking variant of the *onBtnSwitch* system.

6.1.1 Model and Implementation Summary

For testing and evaluation purposes we defined different variants of the *onBtnSwitch* system and used them together with different reconfiguration plans. Table 6.1 shows an overview of the relevant setups for this thesis. The setups result from instantiating the different varation points of the *onBtnSwitch* system. One important varation point is the selection of the component variants. The button controller and LED controller are modeled and implemented with state kinds stateless, ground state and implicit state. For the stateless variants we reuse the component implementations from the implicit-state variants and simply discard their state during reconfigurations. Each variant exists in two equivalent versions 1.0 and 2.0. The distributed embedded application can combine each button controller with each LED controller. However, from the 36 variants not all are relevant as starting configurations, as most often both components are initially used in their version 1.0. If both components have the same state kinds, these variants of the distributed embedded application are called stateless, ground and implicit. In addition, we use a mixed starting configuration, in which we use the components in their version 1.0, but with different state kinds (button controller with implicit state, LED controller with ground state). Besides the component variants, different variants of the topology and deployment are possible. In this evaluation we only use the two-node topology with the corresponding deployments (see Figure 6.2).

Setup	Variant	Reconfiguration
ICSA	stateless*	update both components without state transfer
INDIN-1	ground	update button controller with ground state transfer
INDIN-2	implicit	update LED controller with dump/load and capture/replay
INDIN-3	mixed	consistently add second LED controller on *node1* using state anticipation
Benchmark	mixed	repeatedly update and downgrade both components so that all reconfiguration steps are used many times

TABLE 6.1: How the different variants of *onBtnSwitch* are used in combination with reconfiguration plans. **The ICSA setup reuses the implicit-state implementations.*

Besides the (initial) system variants, the reconfiguration plans are important for the different setups. In this evaluation, we do not trigger the reconfigurations during runtime. Instead, we provide initialization plans which contain both the initial steps required to launch the distributed embedded application and then the instructions for the "dynamic" reconfiguration. Table 6.1 shows five reconfiguration setups based on the four static system variants. The *ICSA* setup uses the stateless variant and updates both components synchronously without state transfer. This setup is based on the original proposal in [TK19a] and its name refers to the corresponding conference. The setup *INDIN-1* is based on the ground-state variant of *onBtnSwitch*, while *INDIN-2* uses the implicit-state variant and *INDIN-3* uses the mixed variant. Together with the corresponding reconfigurations, these setups are based on the original proposal in [TK19b] (the names refer to the corresponding conference, too). *INDIN-1* updates only the button controller transfering its ground state from the old version to the new one during a cycle turnover without blocking. In *INDIN-2* the LED controller is updated. This time, the implicit-state variant is used, so dump/load is needed to transfer the state, which requires some blocking. To patch the broken dataflow during the bounded downtime, the reconfiguration plan also contains capture/replay steps. The more complex setup *INDIN-3* demonstrates how update package components can be used for state transformation. A second LED controller is added on *node1*, so that button presses invert the LEDs of both nodes. To ensure that the LED states are consistent while reducing the amount of temporary blindness, the reconfiguration plan uses the aggregator-anticipator pattern. Finally, the *Benchmark* setup is based on the mixed-state variant of *onBtnSwitch*. The reconfiguration plan uses loop/endLoop to repeatedly update and downgrade the *onBtnSwitch* system. As its purpose is to gather runtime statistics for the agent and all reconfiguration steps, we do not use the most straight-forward plan to perform the "continuous updates". Instead, we add some unnessessary steps in between, so that we can measure runtime statistics for all reconfiguration step kinds. The design-time model is almost completly specified in section 3.1 and not repeated here. The reconfiguration plans are based on the templates described in section 4.3. Appendix A provides the full reconfiguration plans used in each of the setups.

6.1.2 Comparison of Simulation and Prototype

We describe the behavior of the distributed control system for each of the setups (excluding *Benchmark*). We compare the expected behavior according to the model-based analysis and simulation with the real behavior of the running prototype. The focus of this evaluation is to answer whether the prototype properly fulfills M-4

(embedded control by eApp) and M-5 (reconfigure eApp during operation): Do the agent and software components behave according to the formal specification at least to the extent that the real prototype can be analyzed using the model? To answer this question, we compared the simulated execution from the model with the real execution using the simulation trace and the RTCA log. We checked if the same reconfiguration plans could be used in both the model and the prototype to achieve the same system behavior and point out minor differences. For each of these setups we visualize the EDFG and analyze the quality degradation caused by the reconfigurations as described in section 4.2. We describe how the reconfiguration plans can be constructed systematically using the concepts from the blueprints in section 4.3 and case-specific considerations. A general utility of the model is indicated as we can systematically construct and analyze the reconfigurations using the EDFG analysis and successfully run it in the prototype. Special care has to be taken, though, to work around minor differences between the model and implementation.

ICSA

Figure 6.3 visualizes the EDFG and reconfiguration steps of the *ICSA* setup. The full reconfiguration plan can be found in listings A.3 and A.4. A full reconfiguration plan with alternative "straight-forward" timing can be found in listings A.1 and A.2. The alternative timing is indicated by the faded duplicated steps in figure 6.3. Besides a few minor differences, the *ICSA* reconfiguration is an instantiation of the DAG-blueprint described in section 4.3.3. The initial start reconfiguration in the plans instantiate the start blueprint described in section 4.3.4. During the "dynamic" reconfiguration part, the button controller *btn-v1* is replaced by the new version *btn-v2* at cycle offset 0 via the *rmSwc* and *addSwc* steps. At the same time, the button input is re-mapped via the *rmIO* and *addIO* steps. No state transfer is performed in the *ICSA* setup, so the implicit state *btnPrev*, in which the component internally stores the information, whether the button was pressed before, is lost. If the button was kept pressed during the switch, then *btn-v2* would report a wrong *risingFlank* as output. The *risingFlank* output is rewired in the reconfiguration hook before output processing at cycle offset 1 on *node1* using *rmCom*, *rmMsg*, *addMsg*, and *addCom* steps. The re-creation of the network message is required if the quota must be adapted to a changing interface, for instance. The last *risingFlank* from *btn-v1* appears on *node2* during execution phase of cycle offset 0 and is handed to the old LED controller version *led-v1* for the last time in the input processing phase in cycle offset 1. After input processing, the communication is reconfigured on *node2*, so that subsequent *risingFlanks* (from *btn-v2*) are processed by *led-v2*. The first message from *btn-v2* is created during cycle offset 0, released and transmitted to *node2*, and must be processed by *led-v2* from cycle 2 on. Thus, *led-v2* replaces *led-v1* in cycle offset 2 due to the *rmSwc* and *addSwc* steps, together with the I/O-rewiring.

This reconfiguration plan works both in the Maude simulation and the RTCA prototype. The last end-to-end dataflow of the old system version is highlighted in orange, and the first dataflow of the new system version in green. No downtime of the system is introduced in terms of temporary blindness, because in all cycles during the reconfiguration there exists a path from the button sensor to the LED actuator. Additionally, the reaction time of the system is maintained, because the lengths of the shortest paths from the sensor to the actuator is constantly at three periods (excluding I/O sampling delays). In theory, the consistency of the embedded application is at risk by the missing state transfer, if the button is pressed during the update. The reconfiguration designer may decide that in this case the

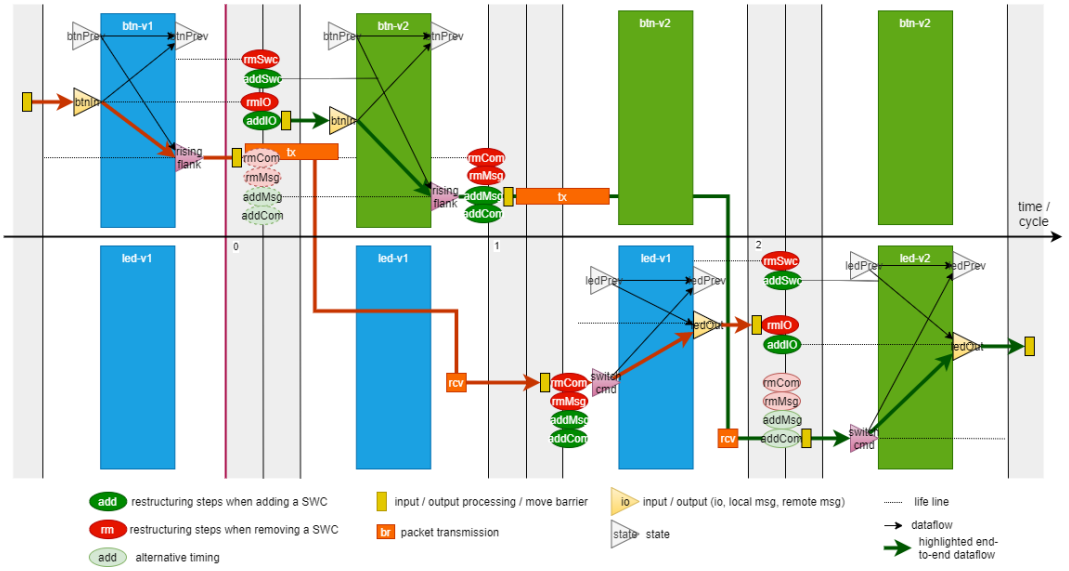

FIGURE 6.3: Combined visualization of an excerpt of the EDFG and the reconfiguration for *onBtnSwitch* in the *ICSA* setup. Both components are updated synchronously without state transfer.

impact is low, if the fault is recognizable and the button can just be pressed once more without a damage. In figure 6.3 some alternative timing options are shown for the communication-related restructuring steps. They correspond to the "straight-forward" timing of the individual component replacements in the minor update blueprint (see section 4.3.2). According to our evaluation runs, this timing works in the prototype, too. However, in our current formal specification, each network message must exist at transmission time, so only the highlighted reconfiguration works in the Maude simulation. This is fixed in the implementation by using the logical step execution time instead of the real execution time.

INDIN-1

Figure 6.4 visualizes the EDFG and reconfiguration steps of the *INDIN-1* setup. The full reconfiguration plan can be found in listings A.5 and A.6. This reconfiguration example uses the "straight-forward" timing of the minor update blueprint (see section 4.3.2) to update the button controller with state transfer. All modification steps are placed in one cycle turnover at cycle offset 0. This requires that the ground state of the button controller is compatible in both versions and small enough, so that it can be transfered under ZET assumption. Additionally, the *risingFlank* output is kept compatible, so that the network message and the LED controller do not need to be reconfigured. This reconfiguration plan works in both the Maude simulation and the RTCA prototype. The last end-to-end dataflow of the old system version and the first dataflow of the new system version are highlighted, again. Additionally, the dataflow caused by the state transfer is highlighted. There is no downtime during the update, as there is always a path from the button input to the LED output. Due to the state transfer, the consistency of the system is maintained in addition. As the state transfer is performed under ZET assumption using the ground state and the *transfer* step, the reconfiguration does not cause an impact on the reaction time. Based on the *INDIN-1* example the ICSA update from the previous section can be extended easily to include a state transfer without causing downtime: If both components have a ground state which can be transfered under ZET assumption,

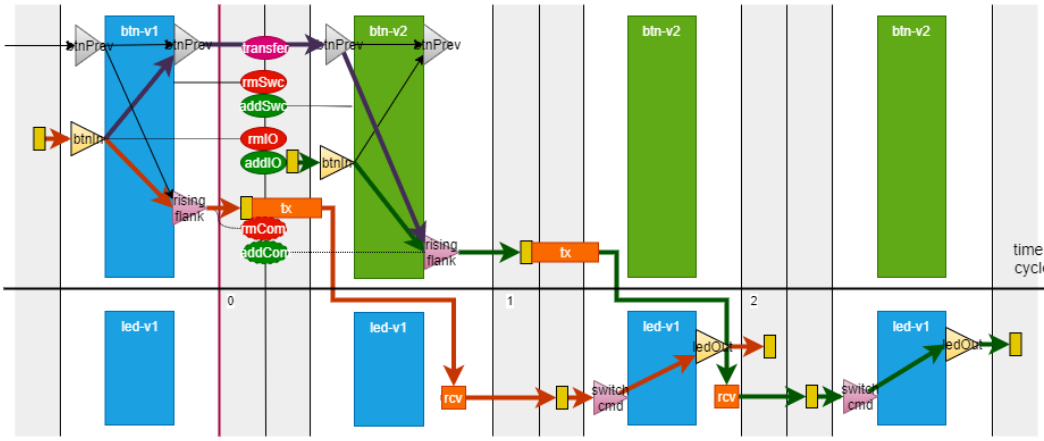

FIGURE 6.4: Combined visualization of an excerpt of the EDFG and the reconfiguration for *onBtnSwitch* in the *INDIN-1* setup. The button controller is replaced with state transfer without downtime.

then the *transfer* step would be used in cycle 0 for the button controller and in cycle offset 2 for the LED controller.

INDIN-2

Figure 6.5 visualizes the EDFG and reconfiguration steps of the *INDIN-2* setup. The full reconfiguration plan can be found in listings A.7 and A.8. In this reconfiguration, the LED controller on *node2* is updated and its implicit state is transferred in three cycles. The red highlighted dataflow shows the last fully working end-to-end data processing and transmission chain of the old system version. The old LED controller version *led-v1* is deactivated in cycle offset 0. As the communication mapping for the remote input is removed from the LED controller on *node2* at cycle offset 0, subsequent *risingFlank* messages from the button controller are dropped. This is transparent to *node1* and thus no reconfiguration step is assigned to *node1*. As the LED controllers have implicit state, at least the time-consuming *dump* and *load* steps are required to transfer the state. In addition, we also run the *transfer* step in the background worker instead of running it under ZET assumption. The dumped LED state would be small enough, but this example demonstrates the transferral in background and the impact on the reconfiguration timing. The duration of each of the three handover steps can be calculated from the modeled state size, quota and WCET, or directly from measurements. For the transfer step we can model an *eta* of 2 ms based on the measurements from the benchmark setup (see subsequent section 6.1.3). Thus, this step takes one period of logical time, if the background worker gets at least 2 ms of CPU quota per period. Memory quota is not covered in the prototype. The *eta* of the *dump* and *load* steps can be modeled with 3 ms based on the benchmark measurements. Due to the quota of 10 ms of the containers, these steps fit into one cycle, too. The resulting timing of the dataflow for the state transfer is highlighted in dark purple. As the *dump* step is triggered in cycle offset 0 and completed by *led-v1* during one period, the dumped state is ready for the *transfer* step in offset 1. The tranferal is completed by the background worker within one period, so the state dump is available for loading at cycle offset 2. The *load* step is triggered and completed in cycle offset 2, so the new LED controller version is ready to work with the transferred and loaded implicit state at cycle offset 3. Due to this state transfer, we thus need three cycle between deactivating *led-v1* in cycle 0 and activating *led-v2*

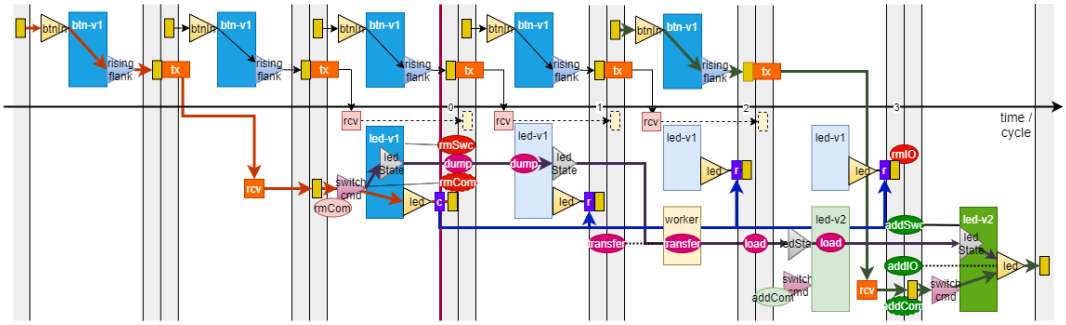

FIGURE 6.5: Combined visualization of an excerpt of the EDFG and the reconfiguration for *onBtnSwitch* in the *INDIN-2* setup. The LED controller is replaced with state transfer including dump/load and capture/replay.

in cycle 3. The first end-to-end dataflow of the new system version is highlighted in green. Finally, we assume that the LED output must be provided every cycle, but there are three cycles in which none of the two versions is working. Thus, we use a *capture* step in cycle offset 0 in the hook *beforeOutputs* to save the last output of the old version. This output is replayed three times during the subsequent cycles to refresh the output during the transition phase. The dataflow resulting from the interception steps is highlighted in blue. The *rmIO* step is performed in cycle 3, which is later than the deactivation of *led-v1* because the output mapping must be present at replay time. The lighter *rmCom* and *addCom* steps shows the alternative timing required by the formal model, in case a reconfiguration of the network message was required, as in the ICSA setup. However, as this is not the case in this setup, the "straight-forward" timing works both in the formal model and the prototype.

This reconfiguration causes a temporary quality degradation of the *onBtnSwitch* system as follows. The blocking approach introduces three cycles of temporary blindness, as there is no path from the button input in cycles -2, -1 and 0 to the LED output. The reaction time is temporarily degraded due to the increasing age of the captured LED output. The normal reaction time is three periods (excluding I/O sampling delays). The path length from the button input to the LED output is 4 periods in cycle offset 1, 5 periods in cycle offset 2, and then reaches a maximum of 6 periods in cycle offset 3. The reaction time is back at 3 periods for the LED output in cycle offset 4, again. With a period of 50 ms, this means that for 150 ms any *risingFlank* would be ignored by the LED controller. In general, this could lead to an inconsistent state of the distributed system, e.g. if there were multiple recievers of the *risingFlank* which would need to have an aligned knowledge about button presses. In this examle, the system operator and reconfiguration designer could decide that this is acceptable, because it is unlikely and recoverable as the user would recognize that the press was missed and could fix this by pressing the button again. However, additional measures beyond the platform scope should be considered, e.g. to inform users that an update is ongoing and how to deal with it.

INDIN-3

Figure 6.6 visualizes the EDFG and reconfiguration steps of the *INDIN-3* setup. The full reconfiguration plan can be found in listings A.9 and A.10. After this reconfiguration, a second instance *led-2* of the LED controller is running on *node1* in addition to *led-1* on *node2* (see green components). Both LEDs must be turned on and off consistently when the button is pressed. Therefore, the state of the old instance must be

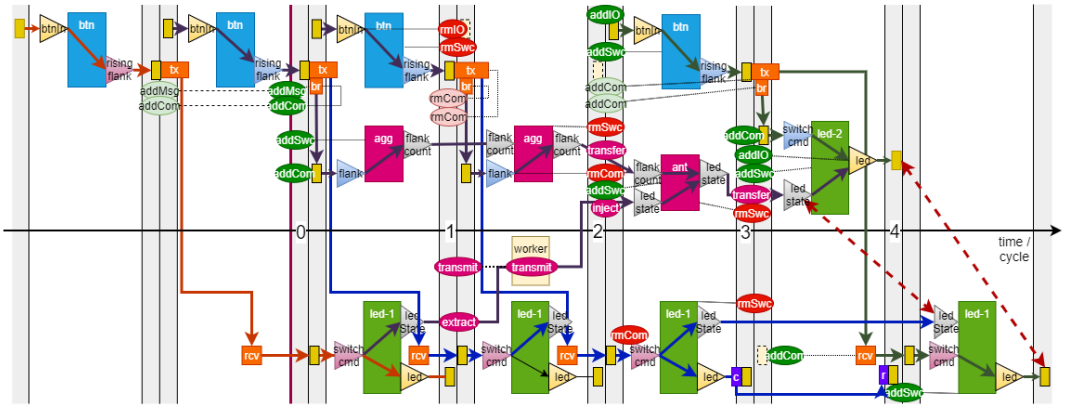

FIGURE 6.6: Combined, partial visualization of the EDFG and the reconfiguration of *onBtnSwitch* in the *INDIN-3* setup. A second LED controller is launched on *node1*. To ensure consistent states of the two LED controller instances at minimal downtime, state transfer and aggregator/anticipator update package components (*agg*/*ant*) are used.

transferred to the new instance. This is achieved by first using an *extract* step under ZET assumption at offset 1 to save the ground state of *led-1*. Immediately afterwards a *transmit* step can be used to send the extracted state to *node1*. The transferal can be assigned an *eta* of 5 ms based on the benchmark measurements (see section 6.1.3). Thus, the state can be transferred within one cycle of the background worker. This state could be injected to *led-2* at offset 2 under ZET assumption to let the new instance start at cycle 2 without any downtime. However, without further measures, if a rising flank of the button occurs during the reconfiguration then the new instance could get an obsolete LED state, because it was extracted before the old instance received the switch command and before the new instance is activated. Consequently, there are 100 ms (offsets -1 and 0, excluding I/O sampling delays) in which the button can be pressed so that the LEDs will have inverted states. This is a situation in which it is better to miss a button input than to have inconsistent system state, as the user can fix the miss by pressing again. The generic solution would be to use transitive blocking during the problematic time span. In this approach, we could determine the required blocking as follows. We would add *led-2* in cycle 2 using the state of *led-1* extracted after cycle 0. Based on that dataflow graph, the switch command would be based on the button input from cycle 1, and the LED state is based on the button input from cycle -2. The blocking approach is to ensure that *led-1* will not get the button input information from cycles -1 and 0, too. This can be achieved by blocking *led-1* at cycle offsets 1 and 2, i.e., two cycles of temporary blindness. In general, if we do not know more about the behavior of the button controller, we should block the whole chain from the sensor input to the actuator outputs along the dataflow graph. In this case, we would block *btn* at cycle offsets -1 and 0, too.

However, we can use the purple aggregator/anticipator components to achive a consistent update at shorter downtime. The aggregator component *agg* is added at cycle offset 0 and removed at offset 2. Its *flank* input is mapped to the *btn* component's *risingFlank* output using *addMsg* and two *addCom* steps – one for mapping the output to the message, and one for mapping the message to the input. The *risingFlank* output is thus mapped to two messages and multicasted to two components. The *agg* component receives both *risingFlank* inputs resulting from the button inputs from cycle -1 and 0, which will be seen by *led-1*, but not by *led-2*. It counts the

positive *risingFlank* inputs and stores the count in its *flankCount* ground state. At cycle offset 2, the state is transfered to the anticipator component *ant* in addition to the LED state received from *node2*. The *ant* component determines the anticipated state of the LED controller based on the fact that two button presses would eliminate each other. The resulting anticipated LED state is produced as ground state and can be transferred to *led-2* in the subsequent cycle 3. Using this aggregator/anticipator pattern we can avoid the temporary blindness for the missed button inputs from cycle -1 and 0. However, due to the anticipator component, we introduce a new blindness of one cycle, which we have to fix. In this dataflow, the *led-2* component receives the *risingFlank* input resulting from the button input at cycle 2, and an anticipated LED state for the button input from cycles until cycle 0. Thus, the button input from cycle 1 must be discarded by blocking *btn* during cycle 1 and *led-2* during cycle 3 due to the communication delay using the shown restructuring steps. Finally, we fix the resulting lack of an LED output using *capture* at cycle offset 3 and *replay* in cycle 4. This setup works both in the model and the prototype. Significant efforts within the barrier qdisc where required to implement multicasting consistently with the logical execution time and reconfiguration approach, but it works and shows that the concepts are feasible even for UDP on Linux. For wiring the aggregator, a working alternative timing of *addMsg* and *addCom* is shown, which better corresponds to the "straight-forward" timing. During the blocking of *btn*, the temporary removal of the communication mapping can be omitted to reduce the number of reconfiguration steps especially in cycle turnover at offset 2. The original publication in [TK19b] was based on an earlier model version in which the removal of communication mappings were assumed implied with *rmSwc*. In the current model, the steps must be present, or the agent will not proceed the simulation due to the missing output. However, the prototype allows communication mappings to be unused, as the agent does not check if a packet is present or not. Instead, a component can send and receive any number of packets until the cycle quota is exceeded and enforced by the barrier.

Some of the important data processing and transmission chains are highlighted, again. The orange dataflow shows the last end-to-end path in the old system version. The green dataflow shows the first normal end-to-end paths in the new system version, now with a multicast to the two LED controllers. The blue and the dark purple dataflow is highlighted as it leads to consistency between the ground states and thus the LED outputs marked with the red dashed lines. Using this approach, the temporary blindness is reduced to one cycle only, compared to the two cycles in the blocking approach. If the state transmission took longer, then the savings would become even bigger, as the aggregation can be pro-longed arbitrarily. The aggregator/anticipator approach induces a temporary blindness, which depends on the WCET and quota of the anticipator component. The blocking approach induces temporary blindness, which depends on the state transmission duration. In this example, as the anticipation can be done in one cycle it is always better than generic blocking in terms of quality degradation. However, this example also raises the question how to handle such "Doppler effect" kinds of reconfiguration problems in general (as we transfer the state in the opposite direction of the dataflow). Regarding the reaction time, a minor quality degradation is caused by the capture/replay phase. The LED output on *node2* in cycle 4 results from the button input in cycle 0, so the reaction time is 4 periods at that time instead of the usual 3 periods (excluding I/O sampling delays). After activating *led-2* in cycle 3, the LED output on *node1* is provided from cycle 4 on with a reaction time of 2 periods. The shorter reaction time regarding the LED output on *node1* results from the logically instant local communication between *btn* and *led-2* in contrast to the remote communication with *led-1*.

6.1.3 Runtime Measurements

We used the *Benchmark* setup to gather runtime statistics on the agent phases (Table 6.2) and the reconfiguration steps (Table 6.3). The full reconfiguration plan can be found in listings A.11, A.12, A.13 and A.14. We ran this setup with the reconfiguration loop for 24 hours. At termination time, the agent had run 863 978 cycles and 8812 full reconfiguration loops on *node1*, and 863 945 cycles and 8811 full reconfiguration loops on *node2*, each loop including a synchronous upgrade and downgrade of both software components and taking approximately 9.8 s. The difference in the numbers of cycles and loops are caused by the initialization behavior prior to the system execution mode (see Section 5.2.3) and the slightly different shutdown timeouts. The divergent sample sizes of the reconfiguration step statistics are additionally caused by the number of occurences of the different steps in the reconfiguration plans for the two nodes. This is because we had to include each reconfiguration step at least once, while keeping the continuous reconfiguration conistent. For instance, we did not use the dump step on *node2*, because the LED controller was configured as ground-state component. The stopwatch component (see Section 5.2.5) was used to measure the runtime statistics based on a start time and an end time for each measurement. The statistics include at least the minimum, maximum, average, and standard deviation of the wall time duration. For some selected measurement hooks, slack and drift statistics were gathered, too, with slack being the time left until the deadline, and drift being the time between the earliest possible start and the real start of the measurement.

agent phase	**min**	**max**	**avg**	**std**	**min**	**max**	**avg**	**std**
		(*node1*)				(*node2*)		
health check	0.018	3.160	1.646	0.244	0.019	3.170	1.614	0.376
before outputs	0.013	1.673	0.116	0.301	0.015	4.245	0.145	0.396
output processing	0.011	1.162	0.482	0.137	0.013	1.571	0.938	0.129
between I/O	0.011	3.470	0.122	0.397	0.011	3.786	0.100	0.393
input processing	0.454	2.233	1.628	0.297	0.440	1.270	0.541	0.103
after inputs	0.012	1.441	0.027	0.117	0.012	1.393	0.055	0.185
trigger	0.018	0.545	0.258	0.061	0.019	0.923	0.283	0.114
sum (no rc)	0.501	7.100	4.014	-	0.491	6.934	3.376	-
sum	0.512	13.684	4.279	-	0.529	16.358	3.676	-
agent overall	0.770	7.448	4.525	0.540	0.708	7.576	3.924	0.686
agent drift	0.446	1.444	0.517	0.028	0.463	1.291	0.528	0.026
agent slack	12.002	18.685	14.959	0.544	11.865	18.792	15.548	0.692

TABLE 6.2: The non-functional behavior of the agent w.r.t. its execution time per cycle in milliseconds, based on a 24 h benchmark run at 100 ms period with 20 ms periodic agent deadline (863 978 cycles on *node1*, 863 945 on *node2*).

Table 6.2 shows the statistics for the different agent phase stages and the overall statistics of the **administration phase**. In the following we first describe the non-reconfiguration stages, then the reconfiguration hooks, and finally the overall agent statistics. The health check took up to 3.170 ms, with an average wall time duration of around 1.6 ms and a standard deviation of 0.244 on *node1* and 0.376 on *node2*. The minimum duration of 0.018 ms was probably measured in cycles in which no software component was in running state, e.g. during system start, which should also apply to the other stages. The output processing phase took up to 1.162 ms (average 0.482 ms) on *node1* for the networked-mapped output of the button controller. On *node2* with the GPIO-mapped output of the LED controller, the output processing

phase took up to 1.571 ms (average 0.938 ms). Thus, GPIO output processing took 0.456 ms longer on average and 0.409 ms in worst-case. However, note that the network output processing only includes the move operation at the barrier qdisc, but not enqueue, peek and dequeue operations. The input processing phase took up to 2.233 ms (average 1.628 ms) on *node1* for the GPIO-mapped input of the button controller. On *node2* with the network-mapped input of the LED controller, the input processing phase took up to 1.270 ms (average 0.541 ms). Again, GPIO input processing took longer both in worst case and on average. Unlike the other stages, the input processing phaes had a noticable minimum duration of 0.440 ms. This is because the single ingress barrier at *ifb0* is moved regardless of whether there is a network-mapped input. This can also explain why the input processing took longer on average and in worst-case. The trigger stage took up to 0.545 ms on *node1* (average 0.258 ms) and 0.923 ms on *node2* (average 0.283 ms).

The statistics for the complete administration phase (including reconfigurations) can be found in the rows "agent overall", "agent drift", and "agent slack". They were taken from the start and end times of each administration phase with the drift as the difference between the start time and the logical execution time in the current cycle $let_c = let_0 + period * c$ and the slack time between the end time and the agent's deadline of 20 ms from let_c. On *node1* the agent's administration phase took between $0.770 - 7.448$ ms (average 4.525 ms, standard deviation 0.540). The maximum on *node2* was slightly higher, at 7.576 ms. The moderate difference between the average and maximum as well as the standard deviation can be explained by the fact that the reconfiguration plan allocates reconfiguration steps "sparsely" to cycles and reconfiguration hooks. The drift of the agent (i.e., the scheduling latency) was between $0.446 - 1.444$ ms, with only few occurences much higher than 0.5 s according to the low average and standard deviation. The observed slack times with respect to the agent's 20 ms deadline was $11.865 - 18.792$ ms. Thus, the agent continuously kept the isochronous tick and deadline during the 24 hours and even would have kept a 10 ms deadline. However, in theory and long-term, the spikes of all the phases could occur in the same cycle. This could lead to a higher overall worst-case execution time of the agent for the *Benchmark* system and reconfiguration. To estimate this, the sums of the agent phase measurements with and without reconfiguration hooks are shown in the rows "sum (no rc)" and "sum". Excluding reconfigurations the maximum sum is 7.100 ms. With reconfiguration hook measurements included, the maximum sum is 16.358 ms. These spikes were not observed during the experiment, but they can indicate the ceiling of the worst-case execution time.

Table 6.3 shows runtime statistics for different **reconfiguration steps**. Note that there are different priorities in our system, as described in Section 5.2.1. Most measurements were done in the agent's cyclic administration phase at critical priority (SCHED_DEADLINE). However, some measurements were taken by different processes at different priorities, according to the responsibilities within the architecture:

- Steps "rcv agent msg" and "rcv response' were done by the agent listener component at elevated priorty (SCHED_FIFO at priority 99) to take messages from the remote agent or the background worker.
- The steps marked with "(bg)" were done by the background worker at normal priority (SCHED_FIFO at priority 42).
- Dump and load were done by the corresponding application components with normal priority, too.

The loop step was done 8813 times on *node1* and endLoop was done 8812 times. Thus, at termination time after 24 h the reconfiguration plan was in the middle of the

step	n	min	max (*node1*)	avg	std	n	min	max (*node2*)	avg	std
loop	8813	0.022	0.028	0.025	0.001	8812	0.022	0.028	0.025	0.001
endLoop	8812	0.110	0.131	0.124	0.003	8811	0.115	0.228	0.127	0.003
notify	17625	0.739	1.099	0.757	0.033	17625	0.735	3.979	0.757	0.041
rcv agent msg	26436	0.093	1.039	0.303	0.293	17626	0.093	0.305	0.105	0.008
wait	70499	0.018	0.063	0.024	0.005	70497	0.019	0.606	0.024	0.006
send cmd	79310	0.166	0.955	0.406	0.203	61683	0.167	1.534	0.259	0.089
rcv response	35250	0.174	6.226	0.295	0.304	35248	0.174	5.104	0.283	0.128
start (in sec)	17626	2.451	2.879	2.616	0.089	17625	2.451	2.877	2.588	0.077
stop (in sec)	17624	0.822	0.864	0.844	0.007	17623	0.736	1.198	0.840	0.007
addSwc	17625	0.017	0.043	0.019	0.001	17624	0.027	0.112	0.030	0.001
rmSwc	17624	0.025	0.393	0.028	0.003	17623	0.019	0.237	0.020	0.002
addCom	26437	0.809	0.939	0.839	0.011	17624	0.769	1.203	0.799	0.015
rmCom	26436	0.744	1.195	0.808	0.057	35247	0.683	1.191	0.729	0.022
addMsg	17625	0.040	0.066	0.045	0.003	17624	0.037	0.349	0.044	0.011
rmMsg	17624	0.033	0.125	0.037	0.001	17624	0.032	0.455	0.036	0.003
addIO	17625	1.253	1.435	1.295	0.014	17624	1.257	1.619	1.388	0.095
rmIO	17624	0.029	0.052	0.040	0.009	17623	0.036	0.139	0.042	0.005
extract (ag)	8812	0.905	1.058	0.924	0.008	0	-	-	-	-
transfer (ag)	0	-	-	-	-	8811	0.942	1.295	0.963	0.014
inject (ag)	8812	0.879	1.035	0.901	0.009	0	-	-	-	-
extract (bg)	0	-	-	-	-	8812	0.982	2.369	1.039	0.072
transfer (bg)	8812	0.975	1.894	1.028	0.036	0	-	-	-	-
inject (bg)	0	-	-	-	-	8812	0.935	2.181	0.960	0.045
transmit (bg)	0	-	-	-	-	8811	1.432	4.457	1.494	0.087
dump btn-v1	8812	1.350	2.238	1.371	0.042	0	-	-	-	-
dump btn-v2	8812	1.348	2.220	1.375	0.065	0	-	-	-	-
load btn-v1	8812	1.479	1.902	1.550	0.028	0	-	-	-	-
load btn-v2	8812	1.461	2.433	1.543	0.053	0	-	-	-	-
capture (io)	8812	0.964	1.249	0.981	0.008	17623	0.952	1.315	0.975	0.013
replay (io)	8812	0.915	1.318	0.936	0.011	44057	1.032	1.459	1.077	0.036
capture (net)	8812	0.434	0.564	0.449	0.008	8812	0.424	0.579	0.453	0.006
stop capt (net)	8812	0.353	0.389	0.367	0.005	8812	0.328	0.462	0.342	0.008
replay (net)	8812	0.382	0.418	0.394	0.005	8812	0.371	0.609	0.385	0.010

TABLE 6.3: The execution times of the reconfiguration steps in milliseconds (seconds for start and stop) as measured during the same 24 h run. The agent had 20 ms/100 ms runtime, and the background worker had 40 % real-time CPU quota.

loop. On *node2*, one loop less was started and completed. The runtime statistics of the loop step are identical for both nodes: It took between 0.022 −0.028 ms, 0.025 ms on average. The endLoop step took between 0.110 −0.228 ms, on average 0.124 ms on *node1* and 0.127 ms on *node2*. The notification step was done 17 625 times on both nodes. It took 0.757 ms on both nodes on average, but on *node2* a much higher spike of 3.979 ms occurred compared to the 1.099 ms maximum measured on *node1*. It is the highest single measurement for reconfiguration operations executed at critical priority. As notification steps are executed in the before output processing, it must have caused the higher spikes there, too (see Table 6.2). The row "rcv agent msg" is a measurement series for when the agent listener processes messages from other agents. At the moment this includes notification messages, transmit messages, and the start time message. The agent listener on *node1* processed 26 436 messages from *node2* (17 625 notification messages and 8811 transmit messages). It took between 0.093 −1.039 ms, 0.303 ms on average. The agent listener on *node2* processed only

17 626 messages from *node2* (17 625 notification messages and the start time message, as *node2* is configured as slave). Here it only took between 0.093 −305 ms, 0.105 ms on average. Thus we can assume that the transmit messages require more processing time than the notification messages, which makes sense because the transmitted state must be written to a new file in the state storage, whereas the notification message only leeds to agent-internal value changes. The wait step was executed 70 499 times on *node1* and 70 497 times on *node2* (approximately eight cycles per loop). As this series covers all three wait step kinds (wait for notification, wait for cycle, wait cycles) it is much closer than expected. However, the wait steps are guarded by their *doesApply()* function, so they are most often only applied and measured once per instance. The wait steps took only 0.024 ms on average on both nodes with a low standard deviation of 0.005 −0.006, but a maximum of 0.606 ms on *node2*. The row "send cmd" is a measurement series for when the agent sends a command to the background worker (start, stop, extract, inject, transfer, and transmit) or to a software component (dump, load). The rows "rcv response" eceive response is a series for when the agent listener receives a message from the background worker in reply to a command (start or stop step completed). On *node1* the agent sent 79 310 command messages: 8812 commands for each of transfer, dump v1, dump v2, load v1, and load v2, plus 17 626 start commands and 17 624 stop commands (two for each loop and one more start during initialization and probably an additional start after which the run terminated). Sending these commands took between 0.166 −0.955 ms, 0.406 ms on average. On *node2* the agent sent only 61 683 command messages: 8811 transfer, 8811 transmit, 8812 extract, 8812 inject, plus 17 625 start and 17 623 stop commands. Sending these commands took between 0.167 −1.534 ms, 0.259 ms on average. Executing the commands extract, transfer, and inject in background took between 0.897 −2.369 ms, close around 1 ms on average. If extract, transfer and inject are directly executed by the agent (rows with "(ag)"), they take about the same time on average, but with much lower spikes only up to 1.295 ms due to the higher priority. Transmit commands in background took longer, between 1.432 −4.457 ms, 1.494 ms on average. Dump and load commands by the software components took between 1.348 −2.433 ms, close to the lower bound on average. Start and stop commands in background took much longer, and are given in seconds. To start the button controller (container start, component initialization, configuration by the agent) took between 2.451 −2.879 s. To start the LED controller took between 2.451 −2.877 s. Stopping the software components took between 0.736 −1.198 s. Thus, start took between 25-29 cycles at the configured period of 100 ms, while stop took between 8-12 cycles. The response messages were processed by that agent listener within 0.283 −0.295 ms on average, but with a few spikes the maximum of which being at 6.226 ms. This was the largest spike within the agent listener and could be caused by the administration phase interrupting the agent listener just after reception of a response message.

The restructuring steps were measured at least 17 623 times, because each upgrade and each downgrade in each loop replace all structural elements to removing the previous software component and adding the new one. The steps *addCom* and *rmCom* were measured once more per loop on *node1*, because the upgrade part of the reconfiguration plan temporarily adds the communication mapping for the replay step. On *node2*, *rmCom* was measured two times more than *addCom*, because removing a remote network-mapped input in the after input processing leads to two rmCom steps: One for marking the step as deleted at the barrier qdisc and one in the subsequent cycle to really delete the mapping (see section 5.2.5). The four

reconfiguration step kinds *addSwc*, *rmSwc*, *addMsg*, and *rmMsg* only lead to agent-internal changes to modify its state and behavior, and thus take only short time: Adding a software component took up to 0.112 ms, but most often much less, with 0.019 −0.030 ms on average and a standard deviation of 0.001. Removing a software component took about the same time on average, but with spikes up to 0.393 ms. Adding and removing a message took between 0.036 −0.045 ms on average. There were only few spikes according to the standard deviations of 0.001 −0.011, but on *node2* they were up to 0.455 ms, while on *node1* the maximum was only at *0.125*ms. To change I/O mappings and communication mappings, the agent must interact with the system, which takes longer. For *addCom* and *rmCom* we measured durations between *0.683*-1.203 ms, with averages around *0.8*ms and standard deviations below *0.06*. In this time the agent sends one or more command messages to the barrier to manipulate the mapping information, also using the message information known by the agent from previous *addMsg* steps, and waits for a positive reply. As already stated above, two separate *rmCom* steps are used in case the mapping info must be preserved after deletion for one more execution phase. For *addIO* to add a GPIO mapping we measured durations between 1.253 −1.619 ms. Adding the LED output on *node2* took a bit longer than the button input on *node1* both on average and at maximum, and seems to have more spikes, according to the standard deviation of 0.095 compared to only 0.014. Finally, the interception steps were measured, devided by type of the output (network-mapped or GPIO-mapped). To capture the GPIO-mapped button input on *node1* took almost the same time as the GPIO-mapped LED output on *node2*, with minimum and average durations close to 1 ms, and spikes of 1.315 ms at maximum. This is expected as both are just files on a *tmpfs* mounted within the containers handled in the same way. The replay steps for GPIO-mapped properties were in the same time range, too, even though "replay (io)" durations were more than 0.1 ms higher on *node2* at minimum, maximum, and on average. Note that on *node2* there were more captures and more replays for the LED as the benchmark reconfiguration plan blocks the LED controller more often than the button controller. Sending and processing capture commands to and by the barrier took between 0.424 −0.579 ms, 0.45 ms on average, with only a small difference between the egress barrier on *node1* and the ingress barrier on *node2*. To send and process stop capture commands (which is always done in the subsequent cycle after a capture command) took 0.34 −0.37 ms on average. A maximum of 0.462 ms was measured on *node2*. Replay steps for network-mapped properties took 0.382 −0.418 ms for egress on *node1* and 0.371 −0.609 ms for ingress on *node2*. The spikes on *node2* might be really higher (not only because of the sample sizes of only 8812) due to the different handling of input and output packets at replay time regarding header manipulation. They are also handled differently at capture time, which however is when the packets are dequeued, not when the capture command is processed.

Table 6.4 shows the runtime statistics for the containerized tasks and handover steps of the **software components**. On the left-hand side, the statistics for the regular task executions are shown. The "old version" (here called "led-v1") of the LED controller ran 453 191 cycles, and the "new version" ran 36 664 cycles (all reconfiguration loops combined). The button controller ran 453 222 cycles in the "old version", and 367 853 cycles in the "new version". The different numbers of cycles result from the imbalanced reconfiguration plan, which runs the "old" distributed application components more often and blocks some components more than others, and from the already mentioned termination shift. In the 719 855 task executions of the two (identical) versions, the LED controller took beetween 1.624 −4.132 ms, 1.68 ms on average. The button controller ran 811 075 task executions in total and took between

series		n	min	max	avg	std
led-v1 (task)	time	453191	1.624	4.132	1.681	0.101
	drift	453191	4.998	8.666	5.295	0.390
	slack	453191	88.639	93.356	93.024	0.449
led-v2 (task)	time	366664	1.624	3.897	1.680	0.100
	drift	366664	4.999	9.880	5.355	0.549
	slack	366664	87.236	93.343	92.965	0.614
btn-v1 (task)	time	453222	0.927	2.555	0.965	0.066
	drift	453222	5.177	8.708	5.517	0.461
	slack	453222	90.090	93.872	93.518	0.482
btn-v2 (task)	time	357853	0.935	2.397	0.968	0.068
	drift	357853	5.191	7.954	5.507	0.387
	slack	357853	90.309	93.859	93.525	0.408

series		n	min	max	avg	std
btn-v1 (dump)	time	8812	1.350	2.238	1.371	0.042
	drift	8812	6.686	8.014	6.810	0.085
	slack	8812	90.421	91.947	91.818	0.095
btn-v1 (load)	time	8812	1.479	1.902	1.550	0.028
	drift	8812	3.158	3.937	3.217	0.044
	slack	8812	94.504	95.348	95.233	0.052
btn-v2 (dump)	time	8812	1.348	2.220	1.375	0.065
	drift	8812	4.991	6.208	5.071	0.050
	slack	8812	92.425	93.640	93.554	0.082
btn-v2 (load)	time	8812	1.461	2.433	1.543	0.053
	drift	8812	7.983	9.366	8.120	0.090
	slack	8812	88.951	90.533	90.337	0.105

TABLE 6.4: The runtime statistics of the software components (time in in milliseconds) in the 24 h benchmark run.

0.927 −2.555 ms, 0.97 ms on average. As the implementations are very similar, it can be assumed that much of this difference is caused by the ground-state handling of the LED controller (read state from file at start, write state at the end). The drift (the latency between the measured start time and the logical cycle start time) of the LED controller was at least 4.998 ms, and took up to 9.880 ms, with an average of approximately 5.3 −5.4 ms. The drift of the button controller had a higher minimum of 5.191 ms, and a higher average of 5.5 ms, but a lower maximum of 8.708 ms. A reason for this could be that the LED controller is input triggered, while the button controller is only triggered by the agent. Thus, the drift of the button controller is primarily caused by the duration of the administration phase at the end of which the agent triggers the task. The LED controller might be triggered earlier by moving the ingress barrier during input processing, but the handling of the input packets could sometimes take longer in the kernel. Another reason might be that the agent on *node1* simply took longer on average, while the agent on *node2* had more and slightly higher peeks, as described earlier. The slack time for software components is the time left between the measured end time of the task execution and the logical start time of the next cycle. In theory, for each cycle and thus for the averages it must hold that $drift + time + slack = period$. Only for the series "btn-v1 (dump)", the sum is 99.999 ms instead of 100 ms, which is probably caused by a floating-point error. The slack time of the LED controller was between 87.236 −93.356 ms, 93.0 ms on average. The button controller left more time on the clock, between 90.090 −93.872 ms, 93.5 ms on average.

On the right-hand side of Table 6.4, the measurements for dump and load are shown. Only the button controller versions ran with implicit state and executed these handover steps once per loop in the reconfiguration plan. Dumping the state took between 1.348 −2.238 ms, on average 1.4 ms. Loading took between 1.461 −2.433 ms, on average 1.5 ms. The higher load times may not be significant or could be caused by temporal correlations in the reconfiguration plan. For instance, during the downgrade part, the load step is performed by *btn-v1* on *node1* in the same cycle in which *node2* transmits the LED controller's state (offset 2 from synchronization point 6). Consequently, the load step is executed in the button controller's container with normal priority, while the agent listener may in parallel already receive the state with elevated priority. The drift and slack statistics look even more

correlated to other steps in the reconfiguration plan. While the drift of the load operation for *btn-v1* was in the lowest range of 3.158 −3.937 ms, for *btn-v2* the drift of the load operation was in the highest range of 7.983 −9.366 ms. An explanation for this is that the agent has five reconfiguration steps scheduled for the cycle turnover leading to loading by *btn-v2*, while only the load step (i.e., sending the command) is scheduled for the cycle turnover before loading by *btn-v1*. The slack time left by dump and load operations was between 88.951 −94.504 ms, which is in a similar range as for the regular tasks.

The benchmark run indicates that the non-functional requirements Q-1 and Q-2 (see Section 5.1.1) may be feasible. All tasks and reconfiguration steps completed in time and it looks like this setup could have run at a much shorter period than the obligatory 100 ms: The longest period of 12.764 ms was observed on *node2*, because the administration phase and the execution phase together left the shortest slack time of 87.236 ms as measured for *led-v2*. However, there are additional time-critical tasks such as reconfiguration steps in background, which took up to 4.457 ms, and the agent listener, which took up to 5.104 ms for a single operation. Additionally, PTP, the network stack, the I/O drivers and the kernel in general need execution time, and parallel software components might interfere with each other. So, it must be evaluated for each specific application and deployment how much shorter periods the current prototype can run stably. This evaluation already shows that significant optimization efforts will be required to reach the stretched goal of 10 ms periods. The long-term goal of 1 ms periods does not seem feasible with this technology stack (hardware, kernel, ...).

6.2 Case Study: *CubeBot*

The *CubeBot* system is a small smart factory showcase system. Its structure is similar to te structure of *onBtnSwitch*, but it uses more realistic interface and runtime technologies. Thus, this case study shows that the platform concepts can be used in more challenging smart factory application scenarios. A similar system called *CamSys* was originally described in [TK19a] as a running example. Meanwhile, we have redesigned the system and implemented the necessary platform extensions and application components as also described in [HLL+22]. We give a brief overview of the *CubeBot* system and describe its design and implementation. Then, we describe the configuration and operation of the *CubeBot* system including its reconfiguration.

6.2.1 System Overview

Figure 6.7 shows the technical context of the *CubeBot* system. A robot takes red and green cubes from a pick area and puts them to one of two place areas to sort them by their colors. There are two versions of this system: A semi-automated and a fully-automated version. During operation of the system, we want to apply our dynamic reconfiguration approach to upgrade from the semi-automated to the fully-automated version. In the semi-automated version, a human worker has to use a button to tell the robot in which of the two place areas the cubes should be placed. Per default the robot continuously picks a cube (or void if no cube is at the pick area) and puts it to the red place area. By pressing the button at picking time, the worker can let the robot put the current cube to the green area. Such a system could be useful for instance if the cubes were heavy, hot, or toxic so that the pick-and-place task must be done by the robot, while the decision is done by the worker. In

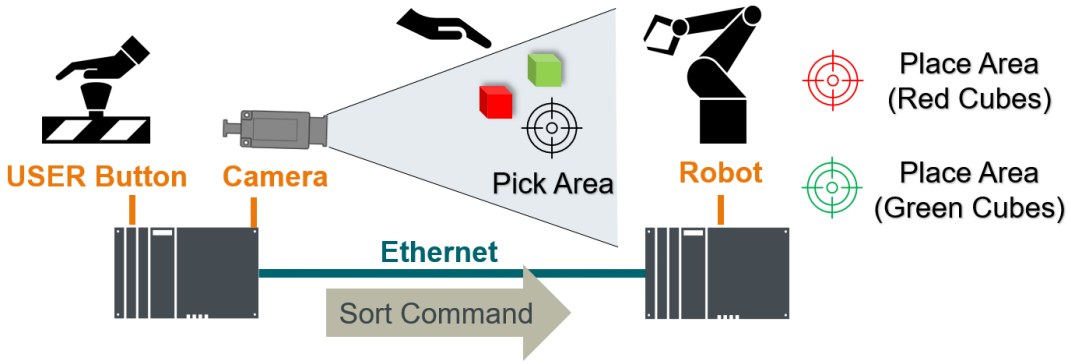

FIGURE 6.7: The *CubeBot* system: A robot sorts cubes by their colors based on input from a worker or from an optical recognition component.

the fully-automated version, a camera view on the pick area is used to automatically detect cubes at the pick area and to recognize their color. In this version, the *CubeBot* can sort the cubes by their colors without a human in the loop. In a real plant there might be some logistic system which automatically moves workpieces to the pick area, where the workpieces are sorted for further processing (e.g. reject workpieces). Additionally, some supervisory control features might be embraced so that wrong decisions by the system could be detected and fixed by a worker. In this showcase system the cubes are put to the pick area by a human, and besides that, the button is the only other means to change the system behavior. To also demonstrate the distributed nature of our system architecture and reconfiguration approach, we are installing the equipment at two controllers and require that the control logic should be provided by application components on both nodes. Even with this limited scope, the *CubeBot* system shows the direction of potential application fields for the real-time container architecture. In a more advanced smart factory, there might be dozens of *CubeBot* systems, which could perform time-critical collaborations based on distributed embedded applications.

6.2.2 Design and Implementation

The *CubeBot* system was designed and implemented during a two-months full-time project of three software engineering master students. Figure 6.8 shows the rough project outline. The real-time container architecture concepts and platform prototype were given as well as the *onBtnSwitch* example application. During an initial onboarding and pre-project phase of two weeks, the students got to know the architectural and operational details. The *CubeBot* system was defined in a few brainstorming sessions as a refinement of the original *CamBot* system and the idea to make a gesture-controlled robot using an already available Lynxmotion 4DOF robotic arm and a usual office webcam. Therefore, the handling of this robot and its serial interface was also part of the pre-project, as well as handling of USB cameras on Linux-based systems. Afterwards, we defined the distributed embedded application and the deployment of the *CubeBot*, which is very similar to the *onBtnSwitch* system. Figure 6.9 shows the detailed architecture for the fully-automated version, which uses a cube detector component for the cube recognition and a robot controller component for sending appropriate commands to the robot. Note that at project execution time, an input barrier was placed at each container instead of one shared barrier at the `ifb0` interface (that was before we supported receiver-side multicasting etc.).

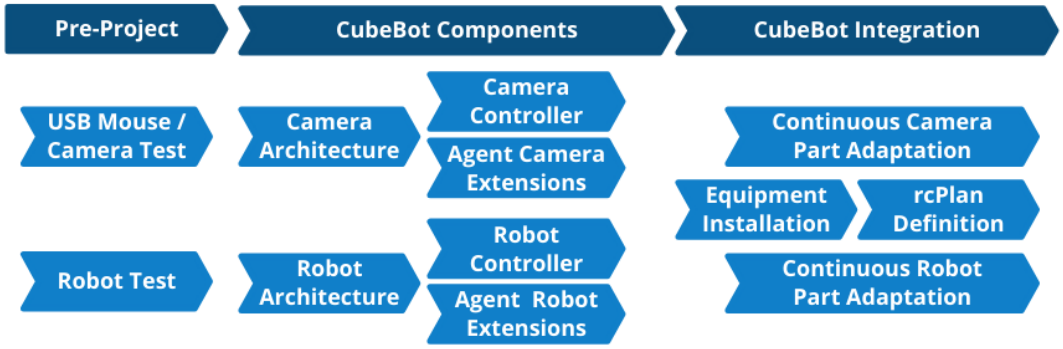

FIGURE 6.8: Overview of the *CubeBot* project showing the rough activities done to design and implement the system.

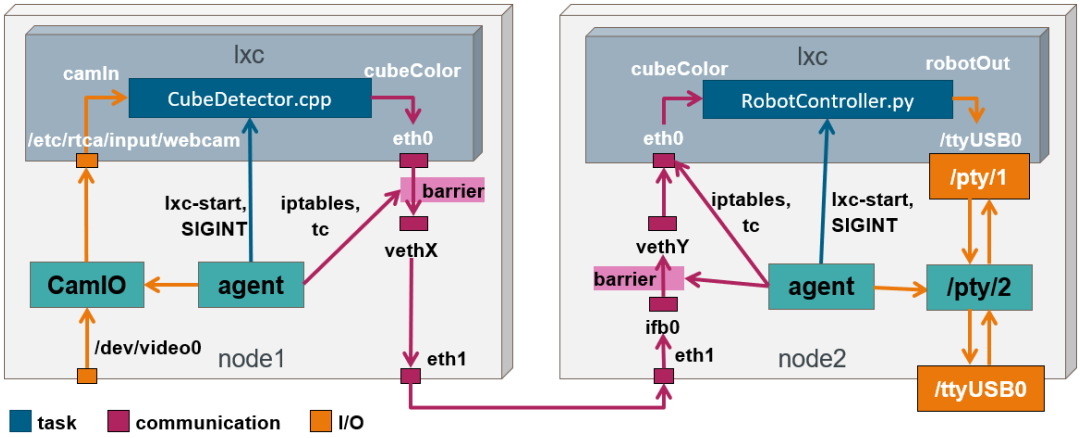

FIGURE 6.9: Detailed architecture of the *CubeBot* system.

The COVID-19 situation enforced a highly separated engineering workflow for the different required design and implementation steps. One separate work stream was the design and development of the **camera extensions** for the real-time container architecture. First of all, the yocto-based Linux image was adapted to offer the video features. The camera input is periodically fetched from the video device by an additional camera worker process and written to a new image file with the current timestamp in a folder accessible in the cube detector's container. At input processing time, the agent updates a symlink within this folder, so that it points to the latest image at that time. In this implementation, the cube detector process could also access older and newer images available in that folder. However, if the component only uses the symlink, this design fulfills the logical execution time paradigm, as the input image is "frozen" during the cycle. This design has the benefit that the input image does not need to be copied – especially not within the time-critical input processing phase. Additionally, the timing with the camera worker is decoupled, so that we do not need any locking and can read the camera in a best-effort manner. The reconfiguration plan description language was extended by new *addCam* and *rmCam* steps, which the agent implements by creating the corresponding folders. The camera worker is started when the agent launches, which was not made re/configurable in the limited scope of the project. A separated activity was the implementation of the stateless cube detector component, which takes a PNG image as input and sends a JSON object with the classification of the image via UDP as output. The classification to red, green and nothing is done via a TensorFlow Lite model generated

from a keras-based convolutional neural network (CNN) trained with 8716 test images. Thus, this work included the definition of the model with Python, the creation of labelled images, the definition of the training procedure and the generation of the C code with TensorFlow Lite, and finally wrapping this executable model in the cube detector component to implement the cyclic classification task. Additionally, a couple of red and green plastic cubes were produced with 3D printers in a maker space at Siemens. The cubes and the *CubeBot* hardware setup were already needed to create the test images for the training.

Another separate work item was the development of the **serial extensions** for the real-time container architecture. The Lynxmotion robot has a serial interface, which can be used via a TTY device in Linux. It is a bi-directional interface, where we can send command messages (paths, waypoints, ...) and request messages (location, ...), some of which are answered by the robot. Thus, the robot is not only an actuator, but also a sensor in a way. One option to handle this would have been to model the robot as two or more I/Os, some of which being inputs and some of which being outputs. Instead, we extended the I/O model, so that also in-out types of equipment are supported, one of which being the additional serial interface. When a corresponding container is started and the I/O mapping is created, the agent also installs pseudoterminal (PTY) devices in the container. For this the reconfiguration plan description language was extended once more by new *addSerial* and *rmSerial* steps. During output processing, the agent reads from the container's PTY and writes everything to the real TTY mapped to it. During input processing, the agent reads from the real TTY and writes everything to the container's PTY. This way, the application component can use the normal serial library, and at the same time, inputs and outputs are both "frozen" during the exectution phase in compliance to the logical execution time paradigm. Due to the serial extensions, the LSS python library from Lynxmotion can be used within the container. Thus, the robot controller component could be developed independently using this library. First, the robot was assembled and mounted to a fixed working area. Then the pick and place commands were defined and tested with the robot without the real-time container architecture integration. For development independently from the robot (e.g. at home), a simple robot simulation component was developed, which mimicks the serial interface of the robot. Finally, the robot control sequence was wrapped in a Python-based software component, to comply with the cyclic task model. For the component-internal design, the state pattern was used. In each cycle, the component state is read from a file, then any inputs are read and the internal knowledge about the cubes and the robot location is updated depending on the inputs and the passed time (calculated from the configured period and a cycle counter). If the current situation (component state, robot state, cube info) requires any action in terms of a command or request, then such output messages are created via the LSS python library and the component state is updated.

After approximately two weeks, the software components where ready in a first version, so the *CubeBot* **integration** phase started. The hardware setup was already done before to create images for the cube detector training and to define and test the coordinates for the robot instructions. The reconfiguration plans for the two nodes were defined and the build and deployment scripts for the integrated project were created. The integrated project pulls all required components from git, builds the components and container images. Afterwards, the exported container images are uploaded to the target nodes and the agents are configured for the *CubeBot* system. As an additional demonstration feature, the unused button of the other node could be used to start a timeboxed system run. This was achieved via a small script, which

continuously reads the button, and if pressed, the agent is started in its current configuration. After the initial integration, the component development activities continued in an iterative way, so that also the integrated system was developed iteratively. Approximately 60 % of the work was done individually at home, and 40 % on-site in the office at the demonstrator. Thus, we estimate that the integration effort was approximately 40 % and up to 60 % were spent in parallelizable and potentially reusable work items. Within the two months the *CubeBot* system was successfully developed and presented at Siemens and at the University of Augsburg.

6.2.3 Configuration and Operation

We describe the final configuration of the real-time container architecture for the successful execution of the *CubeBot* system. This includes the agent configuration files, the container configuration files and the reconfiguration plans. Listing 6.1 shows the agent configuration file for *node1*. The configuration of *node2* is equal except that a different node configuration file is referenced. The global period was set to 500 ms and the agent runtime was set to 50 ms. This was necessary because the cube detector component took up to 300 ms of execution time and the periods must be aligned with each full second in our current prototype due to the initial cycle alignment method (see section 5.2.3). Due to the long period, the agent log level could be set to 7 (debug) to get a full picture of the system execution. A termination trigger of 180 seconds was configured, so there was enough time for demonstrating both versions (semi- and fully-automated) and the seamless reconfiguration. Both *node1* and *node2* were configured as in the *onBtnSwitch* system regarding IP addresses and master/slave roles, so we omit the node configuration files.

```
logLevel: 7
agentRuntimeSeconds: 180
nodeConfigFilePath: "/etc/rtca/robot-deployment/node1.yml"
sched_deadline:
    period_ms: 500
    runtime_ms: 50
    deadline_ms: 500
```

LISTING 6.1: Agent configuration for the *CubeBot* system.

Listing 6.2 shows the configuration of the cube (color) detector's application container *ccd*. The *ccd* container is configured for running on *node1* with its IP address of 192.168.1.101 and the gateway configured to the local virtual bridge *lxcbr0*. A TTY and a pseudo TTY device are configured (even though the pseudo TTY device is not needed for the cube detector component). Finally, we mount the `/etc/rtca/input/webcam` folder into the container. This is where at execution time the camera worker places the webcam images and the agent maintains a symlink according to the camera I/O mapping. A maximum log level is configured for LXC (this is not the application log level, but for the container runtime itself). The remaining configuration items are left unchanged from the busybox template.

Listing 6.3 shows an extract of the robot controller's application container *robot*. This component is configured to run on *node2* with the IP address 192.168.2.100 and the gateway configured to *lxcbr0* on *node2*. The robot controller does need a PTY, which is enabled in the configuration by the corresponding *pts* and *mount* entries. The standard mount configuration for *lib* is commented out, as the robot container is built with a python executable inside. Listings 6.4 and 6.5 show the reconfiguration plans for the *CubeBot* system and the reconfiguration scenario. On *node1* an adapted

```
lxc.loglevel = TRACE
lxc.logfile = ccd.log

lxc.network.type = veth
lxc.network.hwaddr = 00:16:3e:3a:01:02
lxc.network.flags = up
lxc.network.link = lxcbr0
lxc.network.name = eth0
lxc.network.ipv4 = 192.168.1.101/24
lxc.network.ipv4.gateway = 192.168.1.2

lxc.rootfs = /var/lib/lxc/ccd/rootfs
lxc.rootfs.backend = dir
lxc.haltsignal = SIGUSR1
lxc.rebootsignal = SIGTERM
lxc.utsname = ccd
lxc.tty = 1
lxc.pts = 1
lxc.cap.drop = sys_module mac_admin mac_override sys_time

lxc.mount.auto = cgroup:mixed proc:mixed sys:mixed
lxc.mount.entry = shm /dev/shm tmpfs defaults 0 0
lxc.mount.entry = /lib lib none ro,bind 0 0
lxc.mount.entry = /usr/lib usr/lib none ro,bind 0 0
lxc.mount.entry = /sys/kernel/security sys/kernel/security none ro,bind,optional 0 0
lxc.mount.entry = /etc/rtca/input/webcam etc/rtca/input/webcam none ro,bind 0 0
```

LISTING 6.2: LXC configuration file for the cube (color) detector's container *ccd*.

```
lxc.loglevel = TRACE
lxc.logfile = robot.log

lxc.network.type = veth
lxc.network.hwaddr = 00:16:3e:3a:02:01
lxc.network.flags = up
lxc.network.link = lxcbr0
lxc.network.name = eth0
lxc.network.ipv4 = 192.168.2.100/24
lxc.network.ipv4.gateway = 192.168.2.2

lxc.rootfs = /var/lib/lxc/robot/rootfs
lxc.rootfs.backend = dir
lxc.haltsignal = SIGUSR1
lxc.rebootsignal = SIGTERM
lxc.utsname = robot
lxc.tty = 1
lxc.pts = 1
lxc.cap.drop = sys_module mac_admin mac_override sys_time

lxc.mount.auto = cgroup:mixed proc:mixed sys:mixed
lxc.mount.entry = shm /dev/shm tmpfs defaults 0 0
# standard mount commented out to not override python executable
# lxc.mount.entry = /lib lib none ro,bind 0 0
# lxc.mount.entry = /usr/lib usr/lib none ro,bind 0 0
lxc.mount.entry = /sys/kernel/security sys/kernel/security none ro,bind,optional 0 0
lxc.mount.entry = /dev/pts host-pts none rw,bind 0 0
lxc.mount.entry = /var/lib/lxc/robot/rootfs/dev dev none rw,bind 0 0
```

LISTING 6.3: LXC configuration file for the robot controller's container *robot*.

version of the button controller is started first, i.e., the system starts in the semi-automated version. The adapted version is implemented with Python and sends JSON outputs indicating the cube color (default red, green while button is pressed) instead of the *risingFlank*. Thus, it is compatible with the robot controller on *node2* and can be replaced by the cube detector without modifying the robot controller. The robot controller is started on *node2* in the *robot* container using the *addSerial* step. This way it can communicate with the robot plugged at *USB0* indirectly via an agent-managed pseudo TTY configured with baude rate 19 200. Immediately after the initialization reconfiguration, the *ccd* container is started on *node1* with the cube detector component inside. After the component is ready and *50*seconds have passed, the replacement is performed by the second block of modification steps including *addCam*. The *addCam* step makes the camera images available to the inside of the container as symlink called *frame.jpg*. Additionally, the communication mapping is modified, so that the color detector's output are now mapped to *msg1* to reach the robot controller. From this point on, the fully-automated version of the *CubeBot* system is running, so the camera input is used to gather and send cube information to the robot controller.

```
start "button-robot" "192.168.1.103" "/usr/local/bin/python3 /usr/src/app/py-src/main.py"
syncPoint 1 wait "node2"
syncPoint 2 notify "node2"
offset 0:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addSwc "button-robot"
        addGPIO "button-robot" "inputButton" "in" "63"
        addCom "button-robot" "com2" udp src "*:2000" dst "*:2000" via "msg1"
# 'dynamic' reconfiguration plan
start "ccd" "192.168.1.101" "ccd 2>&1"
syncPoint 3 wait 50
offset 1:
    afterInputs:
        rmCom "button-robot" "com2" "msg1"
        addCom "ccd" "com1" udp src "*:2000" dst "*:2000" via "msg1"
offset 2:
    betweenIO:
        rmGPIO "button-robot" "inputButton"
        rmSwc "button-robot"
        addSwc "ccd"
        addCam "ccd" "/frame.jpg"
stop "button-robot"
```

LISTING 6.4: A reconfiguration plan to start an adapted button controller component on *node1* installed as container *button-robot*. After 50 s it is replaced by the cube detector component *ccd*.

```
start "robot" "192.168.2.100" "/usr/local/bin/python3 /usr/src/app/py-src/main.py"
syncPoint 1 notify "node1"
syncPoint 2 wait "node1"
offset 0:
    betweenIO:
        addSwc "robot"
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "robot" "com1" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
        addSerial "robot" "robotSerial" "/dev/ttyUSB0" "19200"
```

LISTING 6.5: A reconfiguration plan to start the robot controller component on *node2* installed as container *robot*.

With these configuration and reconfiguration files, we operated the *CubeBot* showcase system as follows. After pressing the button on *node2*, the system started

and initilly ran in the semi-automated version for three pick-and-place routines of the robot controller. During the semi-automated phase, we placed a red cube in the pick area, which was put to the red place area by the robot. Then we placed a green cube to the pick area and kept the button on *node1* pressed, so the robot put it to the green place area. Afterwards, we let the robot do one routine without a cube to demonstrate that it is unaware of whether or not there are cubes. Shortly before the next routine would start, the system was reconfigured to the fully-automated version, which only starts a pick-and-place routine if a cube is detected. Due to the 50 s reconfiguration timing, the update took place during this empty routine, so the robot stopped at the pick area waiting for a cube. We then waited for a few seconds to demonstrate that the fully-automated version is now active and aware that there is no cube. Afterwards we put some red and green cubes to the pick area one-by-one. They were detected and their color was classified accurately, so the corresponding information was sent to the robot and it picked and placed the cubes to the right place area. In rare cases it took longer to detect a green cube, but in essence the classification worked reliably for the showcase at the end of the project. However, we did not integrate a monitoring system into the demonstrator setup. Thus, a lot of explanation was required to tell the audience what was going on and that this is the expected behavior. We are working on improving the observability, especially during the reconfiguration transition. Hopefully, this will also help to move the focus to the platform concepts rather than the neural network. While we wanted to show that such a technology can be used in the real-time container architecture, the CNN-based application component got too much attraction given our clear target to innovate on the platform level.

Chapter 7

Conclusion

This thesis presented our concepts for modular and dynamically reconfigurable distributed embedded applications. In the introduction in chapter 1 we motivated the need for such concepts by technically enabling smart factories and industrial ecosystem scenarios. Therein, modern IT technology will need to move closer to the technical process, where the control platform needs to maintain determinism, though, so that end-to-end reactivity can be included in the process planning dependably. Additionally, we need to dynamically reconfigure distributed embedded applications due to the use of IT technologies, cloud connectivity, and CPPS adaptation scenarios, while frequent downtime is to be avoided due to cost and also technical reasons. Dynamic reconfigurations should be possible during full operation, i.e., without stopping production, and without the platform being asked to understand the technical process, e.g. to organize production steps so that there are idle times, or wait for such idle times (if any). The reason for this is that a separation of concerns is desired between the distributed embedded application architecture and its dynamic reconfiguration on one hand and the technical process and the organization of the equipment on the other hand. This way any higher-level control concept could fully benefit from our runtime platform concepts transparently and become dynamically reconfigurable. We conclude this thesis by reflecting our key results and pointing out future directions.

7.1 Summary and Discussion

In chapter 2 we described architectural **foundations and existing technical approaches** to industrial control systems. Essential foundations for this thesis include MDE and CBSE, as such concepts from the engineering side strongly contribute to achieving decoupling and deterministic temporal behavior at runtime. At the same time, MDE and CBSE are also targets that are to be technically enabled by our runtime platform concepts for modular and dynamically reconfigurable distributed control systems. We also gave an overview of different architecture styles in distributed systems: while many dynamic concepts rely on service-oriented request/reply communication or event-based publisher/subscriber relationships, our platform concept is based on strictly cyclic sender-receiver communication. We do not allow the applications to dynamically create communication relationships, but strictly orchestrate components from outside, transparently to components. This way we can better achieve end-to-end determinism through static analyis and enforcement (also of timing) at runtime including dynamic reconfigurations, as they are completely in the hand of the runtime platform and we know the exact timing apriory. This is also supported by the time-triggered, isochronous nature of our platform, as we elaborated on when we described different fundamental approaches to industrial

control systems. There are RTOS-based systems along with the problems of schedulability of dependent tasks and coping with priorities, semaphores, and the general complexity resulting from application logic influencing the schedule, for instance through events. Besides the bounded execution time (BET) abstraction provided by RTOSs, the zero-execution-time (ZET) and logical execution time (LET) abstractions exist [KS12]. The LET paradigm is the best abstraction regarding temporal decoupling, i.e., the separation of the concerns of functional control and temporal control, leading to easier integration of distributed (when also applied to communiction) control systems throughout all phases, including dynamic integration at runtime. Thus, the LET paradigm is an essential foundation for our concepts. We also described concrete existing approches to distributed control systems. An important related platform is AUTOSAR, which on the engineering side is very close to our concepts in its classic variant, but not as close on the runtime side, not even in its adaptive variant. Modern IT technologies and dynamic reconfiguration are not supported by AUTOSAR classic (not even easy configuration), while AUTOSAR adaptive does not address end-to-end determinism. We mentioned existing approaches to close these gaps, for example the recent adoption of the LET paradigm in AUTOSAR and the reactor-based solution DEAR [MGLC20] for AUTOSAR adaptive. We explained that reactors may be the event-style sibling to the time-triggered LET concepts and thus to our deterministic aproaches, but that dynamic reconfiguration without downtime is still missing, and that we believe our deterministic concepts better support static analysis. We also gave an overview of related approaches in avionics, for example a harmonization of safety-certified parts on ARINC 653 with POSIX-based IT-style application parts based on hypervisor extensions in ARINC 653 [BS20]. Of course, we also elaborated on existing technogies in automation domains, where we point out the importance of reconfigurable manufacturing systems [KS10], and why we focus on the aspect of reconfiguring the lower-level control software (not the hardware or the higher-level control). If the reader expected an overview on multi-agent systems and the like, we refer to literature cited in the introduction, as self-reconfiguration had to stay beyond the scope of this thesis. We gave an overview of widely used automation technologies such as the IEC 61499 standard for distributed function block diagrams and OPC UA. There is an approach for live updates in PLCs in Cetratus [MM21] based on a switch between primary and secondary "containers" of functionality and including state transferal between them. The I/O virtualization in Cetratus is similar to our concepts, as we also only provide indirect, managed I/O access to the application layer components. However, the approach only updates one local PLC application and does not address distributed dependencies and timing. We highlighted a recent IEC 61499-based approach for dynamic reconfiguration of distributed control systems by Prenzel et al. [PP17], which uses sequences of meta operations available in IEC 61499 in combination with the process update features provided by their Erlang OTP-based implementation. While the extended concept meanwhile even offers rollbacks in case of failures [PHS22], we argued that their approach can cause significant downtime, when function blocks on multiple devices must be suspended during the switch. We also gave a detailed description of embedded Linux, which we built our prototype on. We explain available kernel features which help to achieve hard real-time (especially regarding CPU and network) for distributed, containerized application components in addition to further means that we propose in our concepts. Finally, we described related and foundational formal methods for the specification and analysis of real-time systems and of dynamic reconfiguration. We gave an overview of timed extensions of existing formal frameworks, highlighting timed Kripke structures [LÁÖ15]. They are

used for timed CTL model checking of timed rewrite specifications with Real-Time Maude [Ölv14], which we use to formally specify our concepts. The exisiting formal frameworks for dynamic reconfiguration often ignore real-time requirements, do not target distributed systems, or introduce downtime. Examples for this are the quiescence-based approach by Kamer and Magee [KM90] and the tranquility approach by Vandewoude [VEBD07]. They are still essential concepts and relied on in recent approaches to reconfiguring distributed control systems (with some downtime, though), as by Prenzel and Steinhorst [PS21]. Hammer [Ham09] proposed reconfiguration plans and state transferal in a formal specification based on Maude, which was highly influencial for this thesis, though it does not consider real-time systems. Summing up, besides our work and the work by Prenzel et al., distributed real-time systems are usually only updated during operation when the new versions are backwards-compatible.

Thus, there is a clear gap in the existing work regarding the research scope defined in section 1.1, which we addressed by our concepts. In chapter 3 we describe a **formal framework for decoupled component-based software engineering** for distributed control systems based on Real-Time Maude. The first part of the framework is a design-time model (see section 3.1), the fragments of which need to be instantiated by the different actors to define a distributed control system. The basis is a software component model, which includes ports with required and provided properties, one periodic task, and state declaration (with optional state transfer operations *dump* and *load*), and version information. It is important for later integration steps that also non-functional aspects are modeled, in this case the period and WCET of the task (and state transfer operations, if supported), the supported sample rates or required maximum ages of properties and their sizes (based on their types), and the size limit for the state. Of course, this is a minimal component model compared to others, for example the MARTE component model [OMG19] or the QoS features in DDS [OMG15a]. However, we had to keep this model minimal for the scope of this work to achieve a break-through in later phases, especially regarding the reconfiguration extensions. Further features such as multiple tasks or request/reply interfaces along with more non-functional aspects including the maximum number of parallel requests would make the component model more powerful, but also introduce complexity to the integration concepts. Besides the component description, a component implementation is needed as an independent model fragment, so this can be provided afterwards to implement a component interface, or a description could be created for an existing implementation. The implementation is used for the model-based simulation with Maude, only – it is not intended to generate executable code from it for any given target platform. Based on software component descriptions, distributed embedded applications can be modeled by connecting required and provided properties via their ports. This happens before defining the deployment, similar to the VFB concepts in AUTOSAR [AUT15]. For this, the network topology of the involved MCUs and the MCUs themselves are modeled independently. The model of an MCU includes information about the network ports and I/O ports along with non-functional model information of the throughput of the network ports. Like the software component model, this is a minimalistic model that needs extension in future, but serves as basis for demonstrating the decoupled engineering aspects. The network topology instantiates those MCUs as nodes, connects their network ports via network connection model information, and defines additional non-functional information, in this case a lower bound of the bi-directionally guaranteed throughput. The final design step is to instantiate the deployment model. It maps components to nodes, unmapped properties to I/O ports of the nodes, and connectors

between ports to messages (which themselves are part of the deployment description, too). Thus, each modeling step is quite simple, but still leads to deterministic end-to-end behavior of the distributed control system. For analyzing the feasibility of a given deployment, a few configuration parameters of the runtime model required, such as the reserved time for the agent. The analysis and simulation of a given distributed control system is possible based on a second part of our framework – our runtime model (see section 3.2). For each software component, an application container will be spawned as their sphere of incluence. Containers compete for the system resources CPU and network. When set to running by the scheduler model, runtime up to the modeled WCET or the end of the cycle is used to perform the component's task (or state transfer operation), but not more. The inputs and outputs of a software component declared by its ports are available within the container, but can not be shared or communicated with other components or hardware I/Os. An additional agent is modeled, which runs on each node to provide inputs and handle the outputs during the synchronous cycle turnover on behalf of the components according to the deployment description, for example, create a network message containing the current value of an output. Finally, the runtime model also includes a model of the network behavior, which is similar to the CPU behavior in that messages are competing for transmission time (depending on their sizes) on the medium. Due to the management by the agent, both CPU and network consumption are done according to the LET time paradigm, which leads to end-to-end deterministic behavior without further time slices smaller than the global period. This enables formal analysis to a certain extent as described in section 3.3. Besides statically checking the compatibilities of connected software and I/O ports, end-to-end feasibility can be estimated even without the deployment using a dataflowgraph of a distributed embedded application. After the deployment is defined, this analysis can be refined, in addition to checking that there is enough CPU and network capacity for the allocated tasks and messages. This also takes into account the runtime configuration parameters, such as the configured maximum agent runtime. We also describe how to identify further inconsistencies by running a simulation with Maude, and even how to model check a given distributed embedded application also using the component implementation models and Real-Time Maude, e.g. whether a maximum age constraint of a property can be violated by the architecture. For functionally testing a distributed embedded application in a given deployment, a model of the environment is needed in addition. All in all, this chapter showed how easy integration over all phases is possible, but that the required platform concepts for this are complex even with this simple model and without dynamic reconfiguration.

This framework is extended in chapter 4, so that such distributed control systems can be **reconfigured during full operation**. In three cyclic reconfiguration hooks, the agent checks the top-most reconfiguration step in its node-specific reconfiguration plan and applies corresponding actions if any. We specified various reconfiguration steps which must be combined by a reconfiguration planner to solve a concrete reconfiguration problem. To completely avoid downtime or at least deterministically limit the amount of blocking, we propose and enable following basic reconfiguration schema. Prepare time-critical reconfigurations in background, i.e., by downloading and starting components (*download* and *start*) to be added to the distributed embedded application in dedicated application containers (for instance, new component versions). Such steps are executed in a containerized worker component of the agent at the same priority as application components. Then let all agents involved in a time-critical block of reconfiguration steps agree on a synchronous continuation cycle via the proposed synchronization protocol using the coordination steps

(*notify* and *wait*). The waiting does not block any agent or application parts, but only modifies their behavior in reconfiguration hooks by deferring subsequent reconfiguration steps until that common cycle. At predefined cycle offsets from that continuation cycle, perform the time-critical transition from the old application and deployment to the new one synchronously across nodes using corresponding modification steps. The most essential restructuring steps can change the set of active components (*addSwc* and *rmSwc*), their communication with each other (*addCom*, *rmCom*, *addMsg*, *rmMsg*) and their access to I/Os (*addIO* and *rmIO*). To achieve state transfer between containers, even remote containers, we provide a number of handover steps to handle ground state transfer (*extract*, *transfer*, *transmit*, *inject*) and on-demand dumping and loading of a component's internal task state via the corresponpding optional state transfer operations of the components (*dump* and *load*). As handover steps can take time if the state is big and/or application components are involved (also in transforming the state), the background worker may be asked to perform handover steps in bounded time (by setting *zet* to *false*) and we also provide interception steps to fix potentially broken dataflow timing by saving inputs and outputs and replaying them at later times (*capture* and *replay*). After a synchronous block of reconfiguration steps, clean up the old components by stopping and deleting their containers (*stop* and *delete*) and/or prepare for the next synchronous block.

Obviously, transitioning between applications is the most complex aspect of this approach, as the timing of the reconfiguration steps needs to be aligned so that the end-to-end dataflow from sensors to actuators will be consistent at all times. Additionally, we must not exceed the limits of the resources at any times throughout the reconfiguration, including the runtime requirements of the agent. Thus, we elaborate on the definition and checking of **reconfiguration consistency** in section 4.2. However, while we describe how to calculate the feasibility and check the consistency manually, it is much easier to use a simulation run of the reconfiguration in Real-Time Maude. The simulation will only reach the end of the reconfiguration plan if it is consistent and feasible, as the rules of our model automatically inhibit progression of the system and time, for instance if a component should be added while it is not running, or when a task does not complete its cycle because there is no more CPU time. Thus, a reconfiguration plan is consistent and feasible from platform perspective if and only if the simulation reaches the end of the plan and complete one end-to-end simulation of the resulting distributed embedded application. In addition, we also elaborated on the definition and analysis of the quality degradation caused by a reconfiguration plan with regard to delayed reactions and temporary blindness of the system. For this we proposed the evolving dataflow graph, which is constructed for a given deployment and reconfiguration plan and models each cyclic datapoint and their flow in terms of calculations, transmission, and aging. Its edges are labeled with durations, so that the worst-case reaction time of a system can be defined dynamically as the maximum length of the shortest paths from each sensor to each reachable actuator in the corresponding cycle. Temporary blindness can be defined dynamically as the number of consecutive cycles in which no path exists from a sensor to an actuator. Based on this definition and analysis method, an engineer can see and quantify if there would be a quality degradation and decide whether it is acceptable considering the requireements of the distributed control system and the benefits of the dynamic reconfiguration.

Besides this method for analyzing reconfiguration plans, we also proposed **reconfiguration blueprints** in section 4.3 that help to systematically create consistent reconfiguration plans for some scenarios. The general reconfiguration timing template shows how modification steps can be or should be timed relative to addition

and removal of a software component. If possible, all *add** steps should be done in the hook *betweenIO* just before the first active cycle, and all *rm** steps should be done *betweenIO* just after the last active cycle. Thus, a minor component update can be done by putting all *rm** steps related to the old version and all *add** steps related to the new version into the same cycle turnover. If state transfer is required, we describe how to add the state transfer steps in various cases, the easiest being a compatible transfer of a small enough ground state using the *transfer* step in the same cycle turnover under ZET assumption (if the state is small, the agent can do this within its cyclic reservation, too). Using this basic reconfiguration blueprint, reconfigurations can be achieved without downtime even in case of distributed breaking changes, by synchronously removing and adding components on multiple nodes along the dataflow from sensors through components to actuators. Currently, we only solved this for DAG-style applications and when there is no state or the state is small enough and cyclically maintaind as ground state. However, for starting and stopping applications – even non-DAG applications – we also elaborate on a possibilty to reconfigure cyclic applications. By considering which components are robust against which missing inputs and for which we can provide initial inputs from outside, we can sometimes transform it to a DAG-style application. This approach requires additional knowledge about the application components, though, which is currently not part of our model.

Chapter 5 describes the **real-time container architecture**, a reference implementation of our runtime platorm concepts on Linux, which additionally enables the use of modern IT technolgies. The structure of this chapter is based on the arc42 template for light-weight architecture documentation, which starts with an overview of functional and non-functional requirements. In essence, the platform should enable dynamic orchestration of distributed embedded applications by reconfiguration plans as formally specified in the earlier chapters. As it is an evalutation and demostration prototype, it must also include monitoring and tracing features, especially to gather runtime statistics about the platform and application. The most important quality goals were performance (to achieve 10-100 ms periods), compliant synchronous LET-based real-time behavior (distributed drift less than 1 ms), functional and non-functional isolation of components, and portability (regarding different MCU types and support for different technology stacks of application components). Compatibility with specific neighboring systems was excluded from the platform scope, e.g. certain industrial standards for configuration. The central part was the solution space, were we described the architecture of the runtime platform. The core solution strategy was to use containers on Linux to enforce the isolation of IT-based components and an additional agent to care for the dynamic orchestration according to reconfiguration plans. Existing orchestrators such as kubernetes do not solve this in a time-deterministic way. We described how to still achieve real-time behavior, including configuration of Linux for both PREEMT_RT_FULL (fully preemptive kernel) and RT_GROUP_SCHED (hierarchical real-time scheduling for containers), the use of Linux scheduling policies and priorities (SCHED_DEADLINE for the agent, SCHED_FIFO for the remaining components of the platform and application, except minor priority processes such as ssh), clock and cycle synchronization with PTP and agent functionality, and communication and I/O management by the agent (including the use of a *barrier* Qdisc at the container boarders). The architecture description also provides details on the cyclic agent route, i.e., how input and output processing are done, how the agent triggers the components, and how reconfiguration plans are handled. We described the cyclic task behavior by software components, for which a few restrictions exist so it can work on the real-time container architecture,

for example, cyclically waiting for a signal from the agent. One of the most complex and unique components of our platform is the *barrier* Qdisc, which cares for traffic shaping and dynamic orchestration as configured by the agent. It helps to enforce the LET paradigm for UDP communication transparently to applications, redirects and multi-casts packets according to configured rules, and supports the reconfiguration steps *addMsg*/*rmMsg*, *addCom*/*rmCom*, *capture* and *replay* during the cycle turnover. We explained how this leads to model-compliant traversal of UDP packets between application components. The description also contains details on how the system and individual components are started, which are essential platform tasks. The agent and the containers can be configured by different text-based files. The background worker component of the agent includes a reconfiguration plan parser, so that human-readable, text-based reconfiguration plans can be handed to the agent at runtime. The reconfiguration plan description language (RcPlan DSL) is very similar to the constructor functions for reconfiguration steps in the formal specification in section 4.1. This is important for trusting that the model "faithfully reflects reality" [Gur94], and thus for trusting the simulation-based reconfiguration consistency analysis from section 4.2. The implementations of each reconfiguration step are described – most of them are straight-forward realizations of the modeled behavior in the given architecture, which may have been surprising to the reader. Of course, there are many implementation-specific details not covered in this thesis, but which had to be solved by our runtime platform and were rarely trivial. The prototype description concluded with an overview of complementing systems for development and operation, including the application developement environment, a system monitoring dashboard based on the "ELK" stack, and an update managment system. Some complementing systems were still under construction, but this section shows some first concepts and prototpes how the runtime platform can be integrated with neighboring systems, which is a major task for the future.

In the evaluation in chapter 6 we analysed the feasibility of our engineering and runtime platform concepts using **two case studies**. The first step for the evaluation was already provided by the definition and implementation of the real-time container architecture, as this is needed to show that the formally specified platform concepts can be realized, especially regarding the requirements of synchronous execution on all nodes, LET compliance by applications with IT technologies such as Linux containers and UDP, the ZET assumption for application management and reconfiguration steps, and of course applicability of the reconfiguration steps in a real platform. The feasibility of these model assumptions is the basis for the dynamic orchestration concepts that technically enable the MDE and CBSE vision. For evaluating these aspects, we conducted our first case study based on the ***onBtnSwitch*** system. We described different setups of this system some of which were published in earlier versions: ICSA [TK19a], INDIN-1–3 [TK19b], and Benchmark (not published earlier). The setups use different variants of the components as starting configurations (stateless, ground, implicit, and mixed) and different reconfiguration plans (no state transfer, ground state transfer, state transfer with dump/load, state transformation, and a benchmark upgrade/downgrade loop which uses all steps many times). The ICSA and INDIN setups were modeled and implemented, and we compare the simulation and the real execution in detail. From this comparison we conclude that the prototype implements the modeled platform concepts at least to the extent required by the given setups. That includes synchronous execution across nodes, concurrently starting new containers in background, synchronizing across nodes and then performing time critical modifications before shutting down the old containers in background. In the ICSA setup, a synchronous update of two

distributed components is successfully done without downtime, but also without state transfer. In the INDIN-1 setup, the ground state of the button controller is successfully transferred during the update without downtime. In the more complex INIDN-2 setup, we successfully transfer the implicit state of the LED controller, which participates in the reconfiguration via the *dump* and *load* steps, and we additionally used *capture* and *replay* during the three cycles of downtime required for this state transfer. The INDIN-3 setup was the most complex comparison, but it was also successfully achieved by the prototype. This reconfiguration required multicasting of the *risingFlank* to temporary update package components and remote state transferal via *transmit* in addition to further reconfiguration steps. As calculated from the model, the quality is degraded for one cycle, only, in which the system has a reaction time of four cycles instead of three, and a temporary blindness to the button input is caused for one cycle. To achieve this low-quality degradation despite the remote state transferal, we proposed the aggregator/anticipator pattern [TK19b]. However, we raised the question how "Doppler effect" kinds of reconfiguration problems (where we transfer the state in the opposite direction of the dataflow) can be handled in general. The Benchmark reconfiguration was executed for 24 hours, performing approximately 864 000 cycles with period 100 ms and 8811 full reconfiguration loops. The time-critical cyclic agent routing required a maximum runtime of 7.6 ms, including reconfiguration steps. Together with the maximum observed drift of 1.5 ms, we can conclude that this prototype leaves over 90 % of the cycle to the application in this setup, and that much shorter periods seem feasible. We also concluded that significant optimization efforts will be required to reach 10 ms periods, and that 1 ms periods do not seem feasible with this technology stack. The runtime statistics were much more detailed, though, so they can now be used as indicator for parameterizing the WCETs (*eta*) of the different step kinds in the model to improve the simulation accuracy reagarding the feasibility of a reconfiguration plan. However, we also admitted that these are only preliminary numbers and must be measured for each concrete application, deployment, and reconfiguration plan, as there may be interferences depending on the application types.

In the second case study, the smart factory showcase system ***CubeBot***, we were using more realistic interface and runtime technologies. Additionally, this case study also challenged the CBSE side of our concepts, where our goals were to develop the different application components and required I/O extensions in decoupled activities by different persons. The system was successfully developed within two months by three students und architectural guidance. A large portion of the work was required for the design and implementation of the I/O extensions required to control the equipment according to the platform model (especially isolation, LET, and reconfiguration). For accessing a USB webcam, the camera extensions were introduced to the real-time container architecture prototype. When the camera is added to a container using *addCam* (a specialized version of *addIO*), the best-effort camera images taken by an additional worker component of the agent are cyclically made accessible to the corresponding container as a frozen symlink. For accessing a robotic arm, the serial extensions were introduced. When a serial interface is added to a container using *addSerial* (another version of *addIO*), the agent creates a pair of pseudoterminals, one of which is accessible within the container, and copies in the two directions during input and output processing. Besides an architectural refactoring of the agent, the two extensions were designed and implemented indepenently, and are now available in the prototype. The essential application components were the cube color detector and the robot controller. Both components were developed indepenently using different programming languages (C++ and Python) and the required

frameworks (Tensorflow lite and the Lynxmotion LSS library). Then they were containerized and provided as component for the real-time container architecture. Afterwards, they were integrated by defining a reconfiguration plan for instantiating the corresponding containers on the respective nodes. The only coupled activities during component development were the API definition between the two components – a small YML schema for the UDP payload describing the color of the cube – and between the I/O extensions of the platform prototype. Overall, approximately 40 % of the project time was required for the integration efforts (hardware, platform, interfaces) and up to 60 % were spent in parallelizable and potentially reusable work items such as developing a component. Thus, the *CubeBot* case study showed that the real-time container architecture and the underlying platform concepts are feasible for smart factory technologies and can decouple engineering activities.

All in all, the RTCA is a novel architectural concept towards technically enabling ecosystem scenarios in distributed control systems. The main parts of this thesis are the formal specification and analyzis concepts of a runtime platform and their realization on Linux. The application layer parts of the two case studies were much smaller compared to the amount of effort spent in the platform layer. Nevertheless, we were able to draw conclusions that indicate the feasibility of our concepts and architecture. Future work on the maturity and practical applicability of this promising technology will be required as described in the next section.

7.2 Future Directions

Our novel concepts for modular and dynamically reconfigurable distributed embedded applications will need future work to fully achieve the goals of technically enabling smart factories and industrial ecosystem scenarios. We start with future directions more related to the platform-independent conceptual level of our model. Then we also point out directions more related to the platform-specific level of our prototype. It is a fluent transition, though, as our prototype very closely complies with the model. In the hope to induce much research to finally achieve the ultimate vision, we give a rather comprehensive overview of future directions.

Fundamental concepts

More sophisticated component model: We worked with a minimal component model and minimal non-functional model aspects in our mission to carry this model over all engineering phases until the runtime and dynamic reconfiguration. It is thus an obvious direction to now lift our end-to-end concepts to a more sophisticated component model, potentially based on an industrial standard (e.g. IEC 61499, MARTE, ...). Such a model might for instance support additional communication styles, task models, and non-functional aspects. While events could lead to less determinism (depending on the realization on the different platform layers), request/reply communication would challange the reconfiguration concepts. Another direction is support for multiple tasks of a component, and different periods of tasks, which may make integration and dynamic integration more complex, and also platform implementation. Support for more non-functional aspects might be added such as declaratively modifying communication requirements with “exactly once” and “at least once” modifiers and similar concepts.

More sophisticated system model: We also worked with a minimal system model for the same reasons. In future, the hardware and topology model fragments might need enrichment to better solve hardware heterogeneity problems and support more realistic variety in the network topology. This could include adding more hardware information (e.g. CPU architecture), but also support for more protocols and I/O types. This richer information might be used for model-based platform generation with Yocto, for instance.

Reliability/safety by declaration: As already indicated for the inter-component communication, future work could in general improve the reliability and safety of distributed control systems based on declaratively activatable platform features to be added. Such features could include automatic replication and voting, measures for network reliability such as redundancy, component SDK features for fault tolerant behavior (e.g. send twice), integration with hardware means for more reliability such as a watchdog, and fault handling during reconfiguration (e.g. rollback in-time). Those features and the corresponding fault analysis should be included in the formal specification so that the reachability of hazards can be quantified.

Self-reconfiguration: An application-platform-interface for requesting self-reconfigurations or at least for incluencing reconfigurations should be added. Currently, the reconfiguration timing cannot be modified by the application, which helps to separate the concerns of process control and runtime management. However, it might be required in many cases that the embedded application can temporarily prevent reconfigurations during critical conditions of the technical process, for example. Towards better integrating with common multi-agent systems architectures, it might be required that a privileged application component can use a meta interface to analyze and adapt the distributed embedded application by our reconfiguration means, e.g. to install a new skill, or to build ad-hoc ensembles for real-time collaboration. However, our concepts also create an opportunity for a new adaptability architecture that should be considered in future: That larger-context adaptations of CPPSs are managed by reconfigurations from outside of the distributed embedded applications with more computing power, so that also constrained CPPS can be better utilized.

Reconfiguration plan generation: Currently, the reconfiguration plans are provided to our runtime platform. While we provided blueprints for some cases and analysis methods for checking reconfiguration plans, reconfiguration planning remains a complex task. Future concepts might simplify this task for the other cases by providing blueprints and even automatically deriving plans from more abstract reconfiguration targets.

Concrete prototype improvements

Pull containers: The download step was modeled, but not yet implemented. This should be done in future, as well as integrating a container registry and considering quality gateways. At least checksums and origin should be checked. These concepts should at least be harmonized with the model and not influence the distributed control quality.

More sophisticated applications and technologies: The evaluation case studies were rather small compared to real industrial control systems in factory automation. An evaluation case study with more components and nodes should be defined and implemented. Moreover, more frameworks should be integrated within different reference application components to show how they can be used within the real-time container architecture. This includes alternative execution platforms (e.g. Node-RED, WebAssembly, but also classic runtimes for function block diagrams), middleware (DDS, OPC UA FX, ROS, ...– some will require TCP, support for which will be difficult) and hardware and I/O technologies (GPU, for instance). Does the real-time container architecture need to be integrated with such technologies or can they simply be used within application components? Do some technologies maybe at least require the possibility for nesting runtime responsibilities?

Scalability: The prototype and some of the underlying concepts might need to be more scalable regarding the number of nodes, components, connections, and RAM. For instance, we think that the synchronization protocol might need multicast-notifications or even more advanced means for the case that a huge number of nodes need to be synchronized with potential inter-leaving of synchronous reconfigurations. Also, the scalability of our multi-casting and reconfiguration features within the barrier Qdisc may need to be analyzed and improved.

Performance and maturity: The prototype's performance and maturity is rather low, which may leave doubts about the applicability of the architecture in general. To improve on this, several improvements should be considered, such as using the PREEMPT_RT patches for less OS latency, improving the performance of reconfiguration steps, and offering more performant means for local inter-component communication. More widely used container runtimes and orchestrators might be integrated, e.g. docker and kubernetes, but how to use our re/configuration concepts, then?

Robustness: The current system is not robust against deadline misses of the agent, application components, reconfiguration steps and messages. We need to add concepts towards more stability and/or recovery of the platform. This includes the drift caused by the Linux scheduler when the agent does not progress fast enough and has more reservation left than time until the deadline. Another issue is the reliability of the UDP-based inter-agent communication, which can lead to misses of critical notifications such as the start time. Also, we may need improvements in container and network technologies towards more strict isolation and bandwidth guarantees.

Linux evolution: The current OS image was forked years ago and is outdated, while our customizations will need to be ported to more recent verions. To minimize such efforts in future, the platform should be based on a long-term maintained Linux distribution that enables integration and customization with minimal risk and effort of obsoletion. It should provide means to update the OS- and platform-layer without or with minimal downtime, too. New concepts may be required to achieve full-stack updates without downtime.

Reconfiguration speed: Our concepts avoid or at least minimize downtime during dynamic reconfigurations, but the preparation and cleaning can take "arbitrarily" long. The speed of preparation in background could be improved by

assigning all unused resources to the worker, or using a pool of parallel workers, or by reusing pre-launched elements such as virtual ethernet pair devices and barrier Qdisc instances. Such concepts will make the architecture and reconfigurations more complex, but might be required in application scenarios, for instance, to adapt a robot's skill set fast enough on-demand.

Engineering and operation tools: The engineering side was addressed in the formal framework in this thesis to support decoupling and late integration, but the prototype did not well support these concepts. Future directions should better support the developers and operators. DSL-based validators and generators could reduce complexity via abstraction and automation and reduce the gap between the formal model and prototype. New IDE tools could support the DevOperators by low-code design and dataflow graph visualization. A reconfiguration plan design tool could support in graphical instantiation of reconfiguration templates, visualize a plan's EDFG and the quality degradation, and maybe even derive plans automatically from a "diff" for some cases. Standard cases such as minor component updates could be supported easily by clicking in an update management system or may even support automatic updates. It is not only a usability and configuration management question, though, as we did not solve all reconfiguration problems generically in this thesis. For development and test, integration concepts for (3D) simulations of digital twins with virtual targets for distributed embedded applications would help. Here the problem of simulation time versus real-time must be solved. Finally, improvement on data acquisition and monitoring may be required, as we currently only support remote syslog, which is verbose and has a high and non-deterministic impact on the CPU and network. Additionally, AR/VR-based monitoring tools might be considered in future, which also call for reconfigurability with regard to data acquisition, too.

Security: Unfortunately, we had to exclude security from the scope of this thesis. However, it is clear that security enhancements will be needed in all layers. We already mentioned the inter-agent communication and downloads, which at least requires authentication and integrity checking. Future work should also consider the security of the barrier Qdisc, as this is an essential component within the network stack in the kernel space.

Ecosystem features: In fulfilling the vision of an ecosystem of reusable industrial components and applications, a marketplace will be required. This will include licensing and billing aspects, which may require some features within the runtime platform, for instance license checking, intellectual property protection, and dynamic usage tracking per components. This may be particularly challenging in offline environments.

All in all, we want to apply our concepts to more applications and technologies and make them more usable and applicable. This will hopefully contribute to the smart factory visions of software-defined manufacturing and production as a service. However, non-technical advances will be required, too [BNO+21]. Finally, application scenarios for distributed control systems in other domains might benefit from our concepts as well.

Appendix A

Evaluation Reconfiguration Plans

We provide some of the reconfiguration plans used in the evaluation. For *ICSA* we provide two variants, one which has the "straight-forward" timing, and one which has the originally proposed timing. Corresponding variants exist for all these plans due to the "three-hook equivalence" of restructuring steps, but are omitted.

```
# an executable ICSA RcPlan DSL file for node1
# the most 'straight-forward' timing: rewire everything betweenIO at switch time
# ---------------------------------
# initialization plan
# ---------------------------------
#download "Button Controller" "1.0" "button-v1"
start "button-v1" "192.168.1.101" "button-client 2000" quota 10000

syncPoint 1 wait "node2"
syncPoint 2 notify "node2"

offset 0:
    betweenIO:
        addSwc "button-v1"
        addGPIO "button-v1" "btnIn" "in" "63"
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"

# ---------------------------------
# 'dynamic' reconfiguration plan
# ---------------------------------
# download "Button Controller" "2.0" "button-v2"
start "button-v2" "192.168.1.104" "button-client 2000" quota 10000

syncPoint 3 notify "node2"
syncPoint 4 wait "node2"

offset 0:
    betweenIO:
        rmGPIO "button-v1" "btnIn"
        rmSwc "button-v1"
        rmCom "button-v1" "risingFlank" "msg1"
        rmMsg "msg1"
        addSwc "button-v2"
        addGPIO "button-v2" "btnIn" "in" "63"
        addMsg "msg2" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "button-v2" "risingFlank" udp src "*:2000" dst "*:2000" via "msg2"

stop "button-v1"
# destroy "button-v1"
```

LISTING A.1: Reconfiguration plan for the *ICSA* setup on *node1*.

```
# an executable ICSA RcPlan DSL file for node2
# the most 'straight-forward' timing: rewire everything betweenIO at switch time
# --------------------------------
# initialization plan
# --------------------------------
# download "LED Controller" "1.0" "blink-v1"
start "blink-v1" "192.168.2.100" "blink-server 2000"
syncPoint 1 notify "node1"
syncPoint 2 wait "node1"
offset 2:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
        addSwc "blink-v1"
        addGPIO "blink-v1" "ledOut" "out" "7"
# --------------------------------
# 'dynamic' reconfiguration plan
# --------------------------------
# download "LED Controller" "2.0" "blink-v2"
start "blink-v2" "192.168.2.103" "blink-server 2000"
syncPoint 3 wait "node1"
syncPoint 4 notify "node1"
offset 2:
    betweenIO:
        rmCom "blink-v1" "risingFlank" "msg1"
        rmMsg "msg1"
        rmGPIO "blink-v1" "ledOut"
        rmSwc "blink-v1"
        addMsg "msg2" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "blink-v2" "risingFlank" udp src "*:2000" dst "192.168.2.103:2000" from "msg2"
        addSwc "blink-v2"
        addGPIO "blink-v2" "ledOut" "out" "7"
stop "blink-v1"
# destroy "blink-v2"
```

LISTING A.2: Reconfiguration plan for the *ICSA* setup on *node1*.

```
# an executable ICSA RcPlan DSL file for node1
# as published (but for onBtnSwitch, not CamSys)
# ---------------------------------
# initialization plan
# ---------------------------------
#download "Button Controller" "1.0" "button-v1"
start "button-v1" "192.168.1.101" "button-client 2000" quota 10000
syncPoint 1 wait "node2"
syncPoint 2 notify "node2"
offset 0:
    betweenIO:
        addSwc "button-v1"
        addGPIO "button-v1" "btnIn" "in" "63"
offset 1:
    beforeOutputs:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
# ---------------------------------
# 'dynamic' reconfiguration plan
# ---------------------------------
# download "Button Controller" "2.0" "button-v2"
start "button-v2" "192.168.1.104" "button-client 2000" quota 10000
syncPoint 3 notify "node2"
syncPoint 4 wait "node2"
offset 0:
    betweenIO:
        rmGPIO "button-v1" "btnIn"
        rmSwc "button-v1"
        addSwc "button-v2"
        addGPIO "button-v2" "btnIn" "in" "63"
        rmCom "button-v1" "risingFlank" "msg1"
        rmMsg "msg1"
        addMsg "msg2" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "button-v2" "risingFlank" udp src "*:2000" dst "*:2000" via "msg2"
stop "button-v1"
# destroy "button-v1"
```

LISTING A.3: Reconfiguration plan for the *ICSA* setup on *node1* (timing as published originally).

```
# an executable ICSA RcPlan DSL file for node2
# ---------------------------------
# initialization plan
# ---------------------------------
# download "LED Controller" "1.0" "blink-v1"
start "blink-v1" "192.168.2.100" "blink-server 2000" quota 10000
syncPoint 1 notify "node1"
syncPoint 2 wait "node1"
offset 1:
    afterInputs:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
offset 2:
    betweenIO:
        addSwc "blink-v1"
        addGPIO "blink-v1" "ledOut" "out" "7"
# ---------------------------------
# 'dynamic' reconfiguration plan
# ---------------------------------
# download "LED Controller" "2.0" "blink-v2"
start "blink-v2" "192.168.2.103" "blink-server 2000" quota 10000
syncPoint 3 wait "node1"
syncPoint 4 notify "node1"
offset 1:
    afterInputs:
        rmCom "blink-v1" "risingFlank" "msg1"
        rmMsg "msg1"
        addMsg "msg2" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "blink-v2" "risingFlank" udp src "*:2000" dst "192.168.2.103:2000" from "msg2"
offset 2:
    betweenIO:
        rmGPIO "blink-v1" "ledOut"
        rmSwc "blink-v1"
        addSwc "blink-v2"
        addGPIO "blink-v2" "ledOut" "out" "7"
stop "blink-v1"
# destroy "blink-v2"
```

LISTING A.4: Reconfiguration plan for the *INDIN-1* setup on *node2*.

```
# an executable INDIN-1 (=example 1) RcPlan DSL file for node1
# hooks as published
# --------------------------------
# initialization plan
# --------------------------------
# download "Button Controller" "1.0" "button-v1"
start "button-v1" "192.168.1.101" "button-client 2000 --ground" quota 10000
syncPoint 1 wait "node2"
syncPoint 2 notify "node2"
offset 0:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addSwc "button-v1"
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
        addGPIO "button-v1" "btnIn" "in" "63"
# --------------------------------
# 'dynamic' reconfiguration plan
# --------------------------------
# download "Button Controller" "2.0" "button-v2"
start "button-v2" "192.168.1.104" "button-client 2000 --ground" quota 10000
syncPoint 3 wait 1
offset 0:
    betweenIO:
        rmGPIO "button-v1" "btnIn"
        rmSwc "button-v1"
        rmCom "button-v1" "risingFlank" "msg1"
        transfer "button-v1" "btnStatePrev" "button-v2" "btnStatePrev"
        addSwc "button-v2"
        addCom "button-v2" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
        addGPIO "button-v2" "btnIn" "in" "63"
stop "button-v1"
# destroy "button-v1"
```

LISTING A.5: Reconfiguration plan for the *INDIN-1* setup on *node1*.

```
# an executable INDIN-1 (=example 1) RcPlan DSL file for node2
# reconfiguration only on node1
# --------------------------------
# initialization plan
# --------------------------------
# download "LED Controller" "1.0" "blink-v1"
start "blink-v1" "192.168.2.100" "blink-server 2000" quota 10000
syncPoint 1 notify "node1"
syncPoint 2 wait "node1"
offset 2:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addSwc "blink-v1"
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
        addGPIO "blink-v1" "ledOut" "out" "7"
# --------------------------------
# 'dynamic' reconfiguration plan
# --------------------------------
# done
```

LISTING A.6: Reconfiguration plan for the *INDIN-1* setup on *node2*.

```
# an executable indin-2 RcPlan DSL file for node1
# original published timing
# ---------------------------------
# initialization plan
# ---------------------------------
# download "Button Controller" "1.0" "button-v1"
start "button-v1" "192.168.1.101" "button-client 2000" quota 10000
syncPoint 1 wait "node2"
syncPoint 2 notify "node2"
offset 0:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addSwc "button-v1"
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
        addGPIO "button-v1" "btnIn" "in" "63"
# ---------------------------------
# 'dynamic' reconfiguration plan
# ---------------------------------
# done -- the rest happens on node2
```

LISTING A.7: Reconfiguration plan for the *INDIN-2* setup on *node1*.

```
# an executable indin-2 RcPlan DSL file for node2
# original published timing
# ---------------------------------
# initialization plan
# ---------------------------------
# download "LED Controller" "1.0" "blink-v1"
start "blink-v1" "192.168.2.100" "blink-server 2000" quota 10000
syncPoint 1 notify "node1"
syncPoint 2 wait "node1"
offset 2:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addSwc "blink-v1"
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
        addGPIO "blink-v1" "ledOut" "out" "7"
# ---------------------------------
# 'dynamic' reconfiguration plan
# ---------------------------------
# download "LED Controller" "2.0" "blink-v2"
start "blink-v2" "192.168.2.103" "blink-server 2000" quota 10000
syncPoint 3 wait 1
offset 0:
    beforeOutputs:
        capture "blink-v1" "ledOut" "ledReplay"
    betweenIO:
        rmCom "blink-v1" "risingFlank" "msg1"
        rmSwc "blink-v1"
        dump "blink-v1" "/etc/rtca/tmp/ledStateDump"
offset 1:
    beforeOutputs:
        replay "blink-v1" "ledOut" "ledReplay"
    betweenIO:
        transferBg "blink-v1" "/etc/rtca/tmp/ledStateDump" "blink-v2" "/etc/rtca/tmp/
     ledStateDump"
offset 2:
    beforeOutputs:
        replay "blink-v1" "ledOut" "ledReplay"
        load "blink-v2" "/etc/rtca/tmp/ledStateDump"
offset 3:
    beforeOutputs:
        replay "blink-v1" "ledOut" "ledReplay"
    betweenIO:
        rmGPIO "blink-v1" "ledOut"
        addSwc "blink-v2"
        addCom "blink-v2" "risingFlank" udp src "*:2000" dst "192.168.2.103:2000" from "msg1"
        addGPIO "blink-v2" "ledOut" "out" "7"
stop "blink-v1"
#destroy "blink-v1"
```

LISTING A.8: Reconfiguration plan for the *INDIN-2* setup on *node2*.

```
# an executable INDIN-3 RcPlan DSL file for node1
# timing as originally published
# ---------------------------------
# initialization plan
# ---------------------------------
start "button-v1" "192.168.1.101" "button-client 2000" quota 10000
syncPoint 1 wait "node2"
syncPoint 2 notify "node2"
offset 0:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addSwc "button-v1"
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
        addGPIO "button-v1" "btnIn" "in" "63"
# ---------------------------------
# 'dynamic' reconfiguration plan
# ---------------------------------
# download "LED Controller" "1.0" "blink-v1"
# download "Aggregator" "1.0" "agg"
# download "Anticipator" "1.0" "ant"
start "blink-v1" "192.168.1.100" "blink-server 2000 --ground" quota 10000
start "agg" "192.168.1.106" "agg 2000" quota 5000
start "ant" "192.168.1.107" "ant" quota 30000
syncPoint 3 notify "node2"
syncPoint 4 wait "node2"
offset 0:
    beforeOutputs:
        addMsg "localMsg1" sender "192.168.1.2" receiver "192.168.1.2" quota 10000
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "localMsg1"
        addSwc "agg"
        addCom "agg" "risingFlank" udp src "*:2000" dst "192.168.1.106:2000" from "localMsg1"
offset 1:
    betweenIO:
        rmGPIO "button-v1" "btnIn"
        rmSwc "button-v1"
# these two remove steps below where assumed implied with rmSwc at publication time, but needed
     explicitly meanwhile
# the other option is to remove the addCom steps for button-v1 below at offset 2 when adding
    button-v1, again
# while this is functional, an output might be expected by a stricter agent/barrier (not
    implemented, yet)
#        rmCom "button-v1" "risingFlank" "localMsg1"
#        rmCom "button-v1" "risingFlank" "msg1"
offset 2:
    betweenIO:
        rmCom "agg" "risingFlank" "localMsg1"
        rmSwc "agg"
        transfer "agg" "flankCount" "ant" "flankCount"
        inject "ledStateDump" "ant" "ledState";
        addSwc "ant"
        addGPIO "button-v1" "btnIn" "in" "63"
        addSwc "button-v1"
# on the following two additional addCom steps, see comment above at offset 1
#        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "localMsg1"
#        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
# the following com mapping is one cycle earlier than required, an input might be expected by a
     stricter agent/barrier (not implemented, yet)
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.1.100:2000" from "
    localMsg1"
offset 3:
    betweenIO:
        transfer "ant" "ledState" "blink-v1" "ledStatePrev"
        rmSwc "ant"
        addSwc "blink-v1"
        addGPIO "blink-v1" "ledOut" "out" "7"
# this would be a better place for above addCom, but kept to stick to the published plan
#        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.1.100:2000" from "
    localMsg1"
stop "agg"
stop "ant"
#destroy "agg"
#destroy "ant"
```

LISTING A.9: Reconfiguration plan for the *INDIN-3* setup on *node1*.

```
# an executable INDIN-3 RcPlan DSL file for node2
# timing as originally published
# ---------------------------------
# initialization plan
# ---------------------------------
# download "LED Controller" "1.0" "blink-v1"
start "blink-v1" "192.168.2.100" "blink-server 2000 --ground" quota 10000
syncPoint 1 notify "node1"
syncPoint 2 wait "node1"
offset 2:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addSwc "blink-v1"
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
        addGPIO "blink-v1" "ledOut" "out" "7"
# ---------------------------------
# 'dynamic' reconfiguration plan
# ---------------------------------
syncPoint 3 wait "node1"
syncPoint 4 notify "node1"
offset 1:
    betweenIO:
        extract "blink-v1" "ledStatePrev" "ledStateDump"
        transmitBg "" "ledStateDump" "node1" "ledStateDump"
offset 2:
    afterInputs:
        rmCom "blink-v1" "risingFlank" "msg1"
offset 3:
    beforeOutputs:
        capture "blink-v1" "ledOut" "ledReplay"
    betweenIO:
        rmSwc "blink-v1"
    afterInputs:
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
offset 4:
    beforeOutputs:
        replay "blink-v1" "ledOut" "ledReplay"
    betweenIO:
        addSwc "blink-v1"
```

LISTING A.10: Reconfiguration plan for the *INDIN-3* setup on *node2*.

```
# an executable benchmarking RcPlan DSL file for node1
# we repeatedly update and downgrade the eApp, doing some 'unnecessary' steps, too, for
    gathering runtime statistics
# ---------------------------------
# initialization plan
# ---------------------------------
start "button-v1" "192.168.1.101" "button-client 2000 --silent" quota 10000
syncPoint 1 wait "node2"
syncPoint 2 notify "node2"
offset 0:
    betweenIO:
        addSwc "button-v1"
        addGPIO "button-v1" "btnIn" "in" "63"
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
# ---------------------------------
# 'dynamic' reconfiguration plan
# ---------------------------------
loop 1000000
#
# upgrade
#
start "button-v2" "192.168.1.104" "button-client 2000 --silent" quota 10000
syncPoint 3 notify "node2"
syncPoint 4 wait "node2"
offset 0:
    afterInputs:
        capture "button-v1" "btnIn" "btnReplay"
offset 1:
    beforeOutputs:
        capture "button-v1" "risingFlank" "flankReplay"
    betweenIO:
        replay "button-v1" "btnIn" "btnReplay"
        rmGPIO "button-v1" "btnIn"
        rmSwc "button-v1"
    dump "button-v1" "/etc/rtca/tmp/btnStateDump"
offset 2:
    beforeOutputs:
        rmCom "button-v1" "risingFlank" "msg1"
        rmMsg "msg1"
    betweenIO:
    extract "button-v1" "/etc/rtca/tmp/btnStateDump" "btnStateDump"
    inject "btnStateDump" "button-v2" "/etc/rtca/tmp/btnStateDump"
    load "button-v2" "/etc/rtca/tmp/btnStateDump"
offset 3:
    beforeOutputs:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
        replay "button-v1" "risingFlank" "flankReplay"
offset 4:
    beforeOutputs:
        rmCom "button-v1" "risingFlank" "msg1"
    betweenIO:
        addSwc "button-v2"
        addGPIO "button-v2" "btnIn" "in" "63"
offset 5:
    beforeOutputs:
        addCom "button-v2" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
stop "button-v1"
#destroy "button-v1"
```

LISTING A.11: Reconfiguration plan for the *Benchmark* setup on *node1* (initialization and upgrade part).

```
#
# downgrade
#
start "button-v1" "192.168.1.101" "button-client 2000 --silent" quota 10000
syncPoint 5 notify "node2"
syncPoint 6 wait "node2"
offset 0:
    betweenIO:
        rmGPIO "button-v2" "btnIn"
        rmSwc "button-v2"
        dump "button-v2" "/etc/rtca/tmp/btnStateDump"
offset 1:
    beforeOutputs:
        rmCom "button-v2" "risingFlank" "msg1"
        rmMsg "msg1"
    betweenIO:
        transferBg "button-v2" "/etc/rtca/tmp/btnStateDump" "button-v1" "/etc/rtca/tmp/
     btnStateDump"
offset 2:
    betweenIO:
        load "button-v1" "/etc/rtca/tmp/btnStateDump"
offset 3:
    betweenIO:
        addSwc "button-v1"
        addGPIO "button-v1" "btnIn" "in" "63"
offset 4:
    beforeOutputs:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "button-v1" "risingFlank" udp src "*:2000" dst "*:2000" via "msg1"
stop "button-v2"
#destroy "button-v2"
endLoop syncPoint 3 wait 10
```

LISTING A.12: Reconfiguration plan for the *Benchmark* setup on *node1* (downgrade part).

```
# an executable benchmarking RcPlan DSL file for node2
# with looping reconfigurations for gathering runtime statistics
# --------------------------------
# initialization plan
# --------------------------------
# download "LED Controller" "1.0" "blink-v1"
start "blink-v1" "192.168.2.100" "blink-server 2000 --ground --silent" quota 10000
syncPoint 1 notify "node1"
syncPoint 2 wait "node1"
offset 2:
    betweenIO:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
        addSwc "blink-v1"
        addGPIO "blink-v1" "ledOut" "out" "7"
# --------------------------------
# 'dynamic' reconfiguration plan
# --------------------------------
loop 1000000
#
# upgrade
#
start "blink-v2" "192.168.2.103" "blink-server 2000 --ground --silent" quota 10000
syncPoint 3 wait "node1"
syncPoint 4 notify "node1"
offset 2:
    afterInputs:
        rmCom "blink-v1" "risingFlank" "msg1"
        rmMsg "msg1"
offset 3:
    beforeOutputs:
        capture "blink-v1" "ledOut" "ledReplay"
    betweenIO:
        rmSwc "blink-v1"
        extractBg "blink-v1" "/etc/rtca/tmp/ledStatePrev" "ledStateDump"
    afterInputs:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "blink-v2" "risingFlank" udp src "*:2000" dst "192.168.2.103:2000" from "msg1"
offset 4:
    beforeOutputs:
        replay "blink-v1" "ledOut" "ledReplay"
    betweenIO:
        injectBg "ledStateDump" "blink-v2" "/etc/rtca/tmp/ledStatePrev"
        capture "blink-v2" "risingFlank" "flankReplay"
offset 5:
    beforeOutputs:
        replay "blink-v1" "ledOut" "ledReplay"
    betweenIO:
        rmGPIO "blink-v1" "ledOut"
        addSwc "blink-v2"
        addGPIO "blink-v2" "ledOut" "out" "7"
        replay "blink-v2" "risingFlank" "flankReplay"
stop "blink-v1"
#destroy "blink-v1"
```

LISTING A.13: Reconfiguration plan for the *Benchmark* setup on *node2* (initialization and upgrade part).

```
#
# downgrade
#
# download "LED Controller" "1.0" "blink-v1"
start "blink-v1" "192.168.2.100" "blink-server 2000 --ground --silent" quota 10000
syncPoint 5 wait "node1"
syncPoint 6 notify "node1"
offset 1:
    afterInputs:
        rmCom "blink-v2" "risingFlank" "msg1"
        rmMsg "msg1"
offset 2:
    beforeOutputs:
        capture "blink-v2" "ledOut" "ledReplay"
    betweenIO:
        rmSwc "blink-v2"
        transfer "blink-v2" "/etc/rtca/tmp/ledStatePrev" "blink-v1" "/etc/rtca/tmp/ledStatePrev
     "
        transmitBg "blink-v2" "/etc/rtca/tmp/ledStatePrev" "node1" "ledNode2StateDump"
offset 3:
    beforeOutputs:
        replay "blink-v2" "ledOut" "ledReplay"
offset 4:
    beforeOutputs:
        replay "blink-v2" "ledOut" "ledReplay"
    afterInputs:
        addMsg "msg1" sender "192.168.1.2" receiver "192.168.2.2" quota 10000
        addCom "blink-v1" "risingFlank" udp src "*:2000" dst "192.168.2.100:2000" from "msg1"
offset 5:
    beforeOutputs:
        replay "blink-v2" "ledOut" "ledReplay"
    betweenIO:
        rmGPIO "blink-v2" "ledOut"
        addSwc "blink-v1"
        addGPIO "blink-v1" "ledOut" "out" "7"
stop "blink-v2"
#destroy "blink-v2"
endLoop syncPoint 3 wait 10
```

LISTING A.14: Reconfiguration plan for the *Benchmark* setup on *node2* (downgrade part).

Bibliography

[AB04] Luca Abeni and Giorgio Buttazzo. *Resource reservation in dynamic real-time systems*. In: Real-Time Systems 27.2 (2004), pp. 123–167.

[AB98] Alia Atlas and Azer Bestavros. *Statistical rate monotonic scheduling*. In: Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279). IEEE. 1998, pp. 123–132.

[ABGP05] Colin Atkinson, Christian Bunse, Hans-Gerhard Gross, and Christian Peper. *Component-based software development for embedded systems: an overview of current research trends*. In: Springer, 2005. Chap. Component-based software development for embedded systems: an introduction, pp. 1–7.

[ABH+97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. *Partial-order reduction in symbolic state space exploration*. In: International Conference on Computer Aided Verification. Springer. 1997, pp. 340–351.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. *Model-checking for real-time systems*. In: Proceedings Fifth Annual IEEE Symposium on Logic in Computer Science. IEEE. 1990, pp. 414–425.

[AD94] Rajeev Alur and David L. Dill. *A theory of timed automata*. In: Theoretical computer science 126.2 (1994), pp. 183–235.

[Art22] Artop User Group. *ARText - An AUTOSAR Textual Language Framework*. Internet: `https://www.artop.org/artext/` [Aug 06, 2022]. Artop User Group, 2022.

[Ass13] EAST-ADL Association. *EAST-ADL Domain Model Specification*. Specification 2.1.12. Internet: `https://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf` [Aug 09, 2022]. EAST-ADL Association, 2013.

[AUT15] AUTOSAR. *Virtual Functional Bus*. Specification 4.2.2. Internet: `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_EXP_VFB.pdf` [Aug 09, 2022]. AUTOSAR, 2015.

[AUT16] AUTOSAR. *Classic Platform - Specification of RTE Software*. Specification R19-11. Internet: `https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_SWS_RTE.pdf` [Aug 09, 2022]. AUTOSAR, 2016.

[AUT19a] AUTOSAR. *Classic Platform - Specification of ECU Configuration*. Specification R19-11. Internet: `https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TPS_ECUConfiguration.pdf` [Aug 09, 2022]. AUTOSAR, 2019.

[AUT19b] AUTOSAR. *Classic Platform - Specification of I/O Hardware Abstraction*. Specification R19-11. Internet: `https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_SWS_IOHardwareAbstraction.pdf` [Aug 09, 2022]. AUTOSAR, 2019.

[AUT19c] AUTOSAR. *Classic Platform - Specification of Timing Extensions*. Specification R19-11. Internet: `https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TPS_TimingExtensions.pdf` [Aug 09, 2022]. AUTOSAR, 2019.

[AUT21a] AUTOSAR. *Adaptive Platform - Specification of Communication Management*. Specification R21-11. Internet: `https://www.autosar.org/fileadmin/user_upload/standards/adaptive/21-11/AUTOSAR_SWS_CommunicationManagement.pdf` [Aug 09, 2022]. AUTOSAR, 2021.

[AUT21b] AUTOSAR. *Adaptive Platform - Specification of Execution Management*. Specification R21-11. Internet: `https://www.autosar.org/fileadmin/user_upload/standards/adaptive/21-11/AUTOSAR_SWS_ExecutionManagement.pdf` [Aug 09, 2022]. AUTOSAR, 2021.

[Bar88] Liskov Barbara. *Data abstraction and hierarchy*. In: ACM SIGPLAN Notices 23.5 (1988), pp. 17–34.

[BBK+21] Wolfgang Böhm, Manfred Broy, Cornel Klein, Klaus Pohl, Bernhard Rumpe, and Sebastian Schröck. *Model-based engineering of collaborative embedded systems: Extensions of the spes methodology*. Springer Nature, 2021.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Second. Addison-Wesley Professional, 2003.

[BCPP20] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. *Grand challenges in model-driven engineering: an analysis of the state of the research*. In: Software and Systems Modeling 19.1 (2020), pp. 5–13.

[BD13] Alan Burns and Robert Davis. *Mixed criticality systems-a review*. In: Department of Computer Science, University of York, Tech. Rep (2013), pp. 1–69.

[BD18] Alessandro Biondi and Marco Di Natale. *Achieving predictable multicore execution of automotive applications using the LET paradigm*. In: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE. 2018, pp. 240–250.

[BD91] Bernard Berthomieu and Michel Diaz. *Modeling and verification of time dependent systems using time Petri nets*. In: IEEE transactions on software engineering 17.3 (1991), p. 259.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A tutorial on UPPAAL*. In: Formal methods for the design of real-time systems (2004), pp. 200–236.

[Ber00] Gérard Berry. *The foundations of Esterel*. In: Proof, language, and interaction: essays in honour of Robin Milner. The MIT Press, 2000, pp. 425–454.

[Bir14] Sarah Lynn Bird. *Optimizing resource allocations for dynamic interactive applications*. PhD thesis. University of California, Berkeley, 2014.

[Blo83] Toby Bloom. *Dynamic module replacement in a distributed programming system.* PhD thesis. Massachusetts Institute of Technology, 1983.

[BNO+21] Arne Broering, Christoph Niedermeier, Ioana Olaru, Ulrich Schöpp, Kilian Telschig, and Michael Villnow. *Toward embodied intelligence: Smart things on the rise*. In: IEEE Computer 54.7 (2021), pp. 57–68.

[Bol15] William Bolton. *Programmable logic controllers*. 6th. Newnes, 2015.

[BS20] Gedare Bloom and Joel Sherrill. *Harmonizing ARINC 653 and realtime POSIX for conformance to the FACE technical standard*. In: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC). IEEE. 2020, pp. 98–105.

[BSG18] Gedare Bloom, Joel Sherrill, and Gary Gilliland. *Aligning Deos and RTEMS with the FACE safety base operating system profile*. In: ACM SIGBED Review 15.1 (2018), pp. 15–21.

[Can22] Canonical Ltd. *LXC introduction*. Internet: `https://linuxcontainers.org/lxc/` [Aug 09, 2022]. 2022.

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. *From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications*. In: ACM Sigplan Notices 38.7 (2003), pp. 153–162.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude -A high-performance logical framework: How to specify, program, and verify systems in rewriting logic*. Springer, 2007.

[CDE+16] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.7.1)*. Internet: `http://maude.cs.illinois.edu/w/images/e/e0/Maude-2.7.1-manual.pdf` [Aug 09, 2022]. 2016.

[CFJ+10] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Ramin Tavakoli K., Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Martin Törngren, and Matthias Weber. *The EAST-ADL architecture description language for automotive embedded software*. In: Model-Based Engineering of Embedded Real-Time Systems. Springer. 2010, pp. 297–307.

[Cha17] Ramaswamy Chandramouli. *Security assurance requirements for linux application container deployments*. NIST Interagency/Internal Report (NISTIR) 8176. US Department of Commerce, National Institute of Standards and Technology, 2017.

[Coc20] Richard Cochran. *Linux Programmer's Manual: ptp4l - PTP Boundary/Ordinary Clock*. Internet: `https://linux.die.net/man/8/ptp4l` [Aug 09, 2022]. The Linux Kernel Organization, 2020.

[Cor09] Jonathan Corbet. *Seccomp and sandboxing*. Internet: `https://lwn.net/Articles/332974/` [Aug 09, 2022]. 2009.

[Cor18] Intel Corporation. *libmraa - Low level skeleton library for communication on GNU/Linux platforms*. Internet: `https://iotdk.intel.com/docs/master/mraa/` [Aug 09, 2022]. Intel Corporation, 2018.

[Crn05] Ivica Crnkovic. *Component-based software engineering for embedded systems*. In: Proceedings of the 27th international conference on Software engineering. 2005, pp. 712–713.

[CS21] Luís Carvalho and João Costa Seco. *Deep semantic versioning for evolution and variability*. In: 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP). ACM, 2021, pp. 1–13.

[DDPV21] Dmitrii Drozdov, Victor Dubinin, Sandeep Patil, and Valeriy Vyatkin. *A formal model of IEC 61499-based industrial automation architecture supporting time-aware computations*. In: IEEE Open Journal of the Industrial Electronics Society 2 (2021), pp. 169–183.

[Dij82] Edsger W. Dijkstra. *Selected Writings on Computing: A personal Perspective*. In: Selected Writings on Computing: A personal Perspective. New York, NY: Springer New York, 1982. Chap. On the Role of Scientific Thought, pp. 60–66.

[DKJ18] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. *Multi-agent systems: A survey*. In: IEEE Access 6 (2018), pp. 28573–28593.

[DS95] Jim Davies and Steve Schneider. *A brief history of Timed CSP*. In: Theoretical Computer Science 138.2 (1995), pp. 243–271.

[DV07] Victor Dubinin and Valeriy Vyatkin. *On definition of a formal model for IEC 61499 function blocks*. In: EURASIP Journal on Embedded Systems 2008 (2007), pp. 1–10.

[EAG18] Rolf Ernst, Leonie Ahrendts, and Kai-Björn Gemlau. *System level LET: Mastering cause-effect chains in distributed systems*. In: IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society. IEEE. 2018, pp. 4084–4089.

[EBJ+22] Herve Eychenne, Marc Boucher, Martin Josefsson, Jozsef Kadlecsik, Patrick McHardy, James Morris, Harald Welte, and Rusty Russel. *iptables/ip6tables — administration tool for IPv4/IPv6 packet filtering and NAT*. Internet: `https://ipset.netfilter.org/iptables.man.html` [Aug 08, 2022]. The Linux Kernel Organization, 2022.

[EK10] Drew Eckhardt and Michael Kerrisk. *Linux Programmer's Manual: getrlimit, setrlimit, prlimit - get/set resource limits*. Internet: `https://man7.org/linux/man-pages/man2/getrlimit.2.html` [May 28, 2022]. The Linux Kernel Organization, 2010.

[Eri96] Kelvin T. Erickson. *Programmable logic controllers*. In: IEEE Potentials 15.1 (1996), pp. 14–17.

[Fab76] Robert S. Fabry. *How to design a system in which modules can be changed on the fly*. In: Proceedings of the 2nd International Conference on Software engineering (ICSE). IEEE Computer Society Press, 1976, pp. 470–476.

[FKST23] Joachim Fröhlich, Steffen Klepke, Christoph Stückjürgen, and Kilian Telschig. *Seamless Upgrade: Upgrade functions executing on a control system*. In: Proceedings of the 27th European Conference on Pattern Languages of Programs (EuroPLoP). ACM, 2023.

[FOSS20] Albert Fleischmann, Stefan Oppl, Werner Schmidt, and Christian Stary. *Contextual process digitalization: changing perspectives–design thinking–value-led design*. Springer Nature, 2020.

[Ger09] Rainer Gerhards. *RFC 5424 - the syslog protocol*. Standard. Internet: `https://www.rfc-editor.org/rfc/rfc5424` [Aug 09, 2022]. IETF, Network Working Group, 2009.

[GKC+15] Raul Gorcitz, Emilien Kofman, Thomas Carle, Dumitru Potop-Butucaru, and Robert de Simone. *On the scalability of constraint solving for static/off-line real-time scheduling*. In: International Conference on Formal Modeling and Analysis of Timed Systems. Springer. 2015, pp. 108–123.

[GPR06] Volker Gruhn, Daniel Pieper, and Carsten Röttgers. *MDA®: Effektives Software-Engineering mit UML2® und EclipseTM*. Springer, 2006.

[Gur94] Yuri Gurevich. *Evolving Algebras.* In: Proceedings of the IFIP 13th World Computer Congress. Vol. 1. North Holland, 1994, pp. 423–427.

[Gur95] Yuri Gurevich. *Evolving algebras 1993: Lipari guide*. In: Specification and Validation Methods. Ed. by Egon Börger. Oxford University Press, 1995, pp. 231–243.

[GWC15] Thomas Gaska, Chris Watkin, and Yu Chen. *Integrated modular avionics-past, present, and future*. In: IEEE Aerospace and Electronic Systems Magazine 30.9 (2015), pp. 12–23.

[Ham09] Moritz Hammer. *How to touch a running system: Reconfiguration of stateful components*. PhD thesis. Ludwig-Maximilians-Universität München, 2009.

[HCRP91] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. *The synchronous data flow programming language LUSTRE*. In: Proc. IEEE 79.9 (1991), pp. 1305–1320.

[HDK+17] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. *Communication centric design in complex automotive embedded systems*. In: 29th Euromicro Conference on Real-Time Systems (ECRTS 2017). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:20.

[HGM+22] Bert Hubert, Thomas Graf, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B. Schröder, Jasper Spaans, and Pedro Larroy. *Linux Advanced Routing & Traffic Control HOWTO*. Internet: `https://lartc.org/howto/` [Aug 08, 2022]. Bert Hubert, 2022.

[HHK03] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. *Giotto: a time-triggered language for embedded programming*. In: Proc. IEEE 91.1 (2003), pp. 84–99.

[Hin13] Pieter Hintjens. *ZeroMQ: messaging for many applications*. Internet: `https://zguide.zeromq.org/` [Aug 09, 2022]. O'Reilly Media, Inc., 2013.

[HK21] Serge Hallyn and Michael Kerrisk. *Linux Programmer's Manual: cgroups - Linux Control Groups*. Internet: `http://man7.org/linux/man-pages/man7/cgroups.7.html` [Aug 09, 2022]. The Linux Kernel Organization, 2021.

[HKMM02] Thomas A Henzinger, Christoph M Kirsch, Rupak Majumdar, and Slobodan Matic. *Time-safety checking for embedded programs*. In: EMSOFT. Vol. 2. Springer. 2002, pp. 76–92.

[HKMS09] Thomas A. Henzinger, Christoph M Kirsch, Eduardo R. B. Marques, and Ana Sokolova. *Distributed, modular HTL*. In: 30th IEEE Real-Time Systems Symp. IEEE, 2009, pp. 171–180.

[HLL+22] Joseph Hirsch, Marius Lichtblau, Marian Lingsch Rosenfeld, Kilian Telschig, and Alexander Knapp. *Cube Bot – A Smart Factory Showcase for the Real-Time Container Architecture*. In: 2022 IEEE International Conference on Industrial Informatics (INDIN'22). IEEE. 2022.

[HLR16] Axel Habermaier, Johannes Leupolz, and Wolfgang Reif. *Unified simulation, visualization, and formal analysis of safety-critical systems with*. In: Critical Systems: Formal Methods and Automated Verification. Springer, 2016, pp. 150–167.

[Hoa78] Charles Antony Richard Hoare. *Communicating sequential processes*. In: Communications of the ACM 21.8 (1978), pp. 666–677.

[HR83] Theo Haerder and Andreas Reuter. *Principles of Transaction-oriented Database Recovery*. In: ACM Comput. Surv. 15.4 (1983), pp. 287–317.

[HW04] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[ICS17] ICS. *IEEE 1003.1 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)) Base Specifications*. Standard. IEEE Computer Society, 2017.

[ICS18] ICS. *IEEE 802.1Q - IEEE Standard for Local and Metropolitan Area Networks - Bridges and Bridged Networks*. Standard. IEEE Computer Society, 2018.

[IEC12] IEC. *IEC 61499-1:2012. Function blocks - Part 1: Architecture*. Standard. International Electrotechnical Commission, 2012.

[IEC13] IEC. *IEC 61131-3:2013 - Programmable controllers - Part 3: Programming languages*. Standard. International Electrotechnical Commission, 2013.

[IEC20] TS EN IEC. *IEC 62769-103-4:202 - Field Device Integration (FDI) - Part 103-4: Profiles - PROFINET*. Standard. International Electrotechnical Commission, 2020.

[IMS19] IMS. *IEEE 1588 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Standard. IEEE Instrumentation and Measurement Society, 2019.

[ISO05a] ISO. *ISO 17356-3:2005 - Road vehicles - Open interface for embedded automotive applications - Part 3: OSEK/VDX Operating System (OS)*. Standard. ISO/TC 22 Straßenfahrzeuge, 2005.

[ISO05b] ISO. *ISO 17356-4:2005 - Road vehicles - Open interface for embedded automotive applications - Part 4: OSEK/VDX Communication (COM)*. Standard. ISO/TC 22 Straßenfahrzeuge, 2005.

[ISO13] ISO/IEC. *IEC 62264-1:2013. Enterprise-control system integration – Part 1: Models and terminology*. Standard. CENELEC Europäisches Komitee für Elektrotechnische Normung, 2013.

[ISO18] ISO/IEC. *ISO/IEC 9899:2018 - Information technology – Programming languages – C*. Standard. International Organization for Standardization, 2018.

[ISO19] ISO/IEC. *IEC 15909-1:2019. Systems and software engineering — High-level Petri nets — Part 1: Concepts, definitions and graphical notation*. Standard. International Electrotechnical Commission, 2019.

[ISO20] ISO/IEC. *IEC 62541-1:2020. OPC unified architecture - Part 1: Overview and concepts*. Standard. International Electrotechnical Commission, 2020.

[Jan10] Stephan Janisch. *Behaviour and refinement of port-based components with synchronous and asynchronous communication*. PhD thesis. Ludwig-Maximilians-Universität München, 2010.

[Jan19] Jan Engelhardt. *Packet flow in Netfilter and General Networking*. [Internet: `https://de.wikipedia.org/wiki/Datei:Netfilter-packet-flow.svg` [Aug 08, 2022]. 2019.

[KC03] Jeffrey O. Kephart and David M. Chess. *The vision of autonomic computing*. In: IEEE Computer 36.1 (2003), pp. 41–50.

[Ker02] Michael Kerrisk. *Linux Programmer's Manual: capabilities - overview of Linux capabilities*. Internet: `https://man7.org/linux/man-pages/man7/capabilities.7.html` [Aug 09, 2022]. The Linux Kernel Organization, 2002.

[Ker16] Michael Kerrisk. *Linux Programmer's Manual: tmpfs - a virtual memory filesystem*. Internet: https://man7.org/linux/man-pages/man5/tmpfs.5.html [June 06, 2022]. The Linux Kernel Organization, 2016.

[Ker21] Michael Kerrisk. *Linux Programmer's Manual: sysfs - a filesystem for exporting kernel objects*. Internet: `https://man7.org/linux/man-pages/man5/sysfs.5.html` [Aug 09, 2022]. The Linux Kernel Organization, 2021.

[KLMS11] Christoph M. Kirsch, Luis Lopes, Eduardo R.B. Marques, and Ana Sokolova. *Runtime programming through model-preserving, scalable runtime patches*. In: 2011 Eleventh International Conference on Application of Concurrency to System Design. IEEE, 2011, pp. 77–86.

[KM90] Jeff Kramer and Jeff Magee. *The evolving philosophers problem: Dynamic change management*. In: IEEE Transactions on software engineering 16.11 (1990), pp. 1293–1306.

[Kop11] Hermann Kopetz. *Real-Time Systems*. Second. Springer, 2011.

[KS10] Yoram Koren and Moshe Shpitalni. *Design of reconfigurable manufacturing systems*. In: Journal of manufacturing systems 29.4 (2010), pp. 130–141.

[KS12] Christoph M Kirsch and Ana Sokolova. *The logical execution time paradigm*. In: Advances in Real-Time Systems. Springer, 2012, pp. 103–120.

[KZL+22] Michael Kerrisk, Peter Zijlstra, Juri Lelli, Tom Bjorkholm, Markus Kuhn, and David A. Wheeler. *Linux Programmer's Manual: sched - overview of CPU scheduling*. Internet: `http://man7.org/linux/man-pages/man7/sched.7.html` [Aug 09, 2022]. The Linux Kernel Organization, 2022.

[LÁÖ15] Daniela Lepri, Erika Ábrahám, and Peter Csaba Ölveczky. *Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories*. In: Science of Computer Programming 99 (2015), pp. 128–192.

[Lee21] Edward A Lee. *Determinism*. In: ACM Transactions on Embedded Computing Systems (TECS) 20.5 (2021), pp. 1–34.

[LL73] Chung Laung Liu and James W Layland. *Scheduling algorithms for multiprogramming in a hard-real-time environment*. In: Journal of the ACM (JACM) 20.1 (1973), pp. 46–61.

[LMC+20] Nuno Lopes, Rolando Martins, Manuel Eduardo Correia, Sérgio Serrano, and Francisco Nunes. *Container Hardening Through Automated Seccomp Profiling*. In: Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds. 2020, pp. 31–36.

[LMS02] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. *On model checking durational Kripke structures*. In: International Conference on Foundations of Software Science and Computation Structures. Springer. 2002, pp. 264–279.

[LS19] Yannis Lilis and Anthony Savidis. *A survey of metaprogramming languages*. In: ACM Computing Surveys (CSUR) 52.6 (2019), pp. 1–39.

[LSD89] John Lehoczky, Lui Sha, and Yuqin Ding. *The rate monotonic scheduling algorithm: Exact characterization and average case behavior*. In: RTSS. Vol. 89. 1989, pp. 166–171.

[MBW+18] Somayeh Malakuti, Jürgen Bock, Michael Weser, Pierre Venet, Patrick Zimmermann, Mathias Wiegand, Julian Grothoff, Constantin Wagner, and Andreas Bayha. *Challenges in skill-based engineering of industrial automation systems*. In: 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA). Vol. 1. IEEE. 2018, pp. 67–74.

[Mea55] George H Mealy. *A method for synthesizing sequential circuits*. In: The Bell System Technical Journal 34.5 (1955), pp. 1045–1079.

[Mer14] Dirk Merkel. *Docker: Lightweight Linux containers for consistent development and deployment*. In: Linux Journal 239 (2014).

[Mes92] José Meseguer. *Conditional rewriting logic as a unified model of concurrency*. In: Theoretical computer science 96.1 (1992), pp. 73–155.

[MF76] P Merlin and DJ Faber. *Recoverability of communication protocols*. In: IEEE Transactions on Communication 24.9 (1976), pp. 1036–1043.

[MGLC20] Christian Menard, Andrés Goens, Marten Lohstroh, and Jeronimo Castrillon. *Achieving determinism in adaptive AUTOSAR*. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE. 2020, pp. 822–827.

[MM21] Imanol Mugarza and Juan Carlos Mugarza. *Cetratus: Live updates in urogrammable logic controllers*. In: 2021 IEEE International Workshop of Electronics, Control, Measurement, Signals and their application to Mechatronics (ECMSM). IEEE. 2021, pp. 1–7.

[Mon14] László Monostori. *Cyber-physical production systems: Roots, expectations and R&D challenges*. In: Procedia CIRP 17 (2014), pp. 9–13.

[MT00] Nenad Medvidovic and Richard N Taylor. *A classification and comparison framework for software architecture description languages*. In: IEEE Transactions on software engineering 26.1 (2000), pp. 70–93.

[MZ16] Pieter J. Mosterman and Justyna Zander. *Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems*. In: Software and Systems Modeling (SoSyM) 15 (2016), pp. 5–16.

[OD08] Ernst-Rüdiger Olderog and Henning Dierks. *Real-time systems: formal specification and automatic verification*. Cambridge University Press, 2008.

[OEHK07] Roman Obermaisser, Christian El-Salloum, Bernhard Huber, and Hermann Kopetz. *Modeling and verification of distributed real-time systems using periodic finite state machines*. In: Computer Systems Science and Engineering 22.6 (2007), p. 333.

[Old98] E-R Olderog. *Formal methods in real-time systems*. In: Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No. 98EX168). IEEE. 1998, pp. 254–263.

[Ölv14] Peter Csaba Ölveczky. *Real-Time Maude and its applications*. In: International Workshop on Rewriting Logic and its Applications. Springer. 2014, pp. 42–79.

[ÖM02] Peter Csaba Ölveczky and José Meseguer. *Specification of real-time and hybrid systems in rewriting logic*. In: Theoretical Computer Science 285.2 (2002), pp. 359–405.

[ÖM07] Peter Csaba Ölveczky and José Meseguer. *Semantics and pragmatics of Real-Time Maude*. In: Higher-order and symbolic computation 20.1-2 (2007), pp. 161–196.

[OMG12] OMG. *Common Object Request Broker Architecture (CORBA)*. Specification 3.3. Internet: `https://www.omg.org/spec/CORBA/3.3/` [Aug 09, 2022]. Object Management Group, Inc, 2012.

[OMG14] OMG. *Model Driven Architecture (MDA) – MDA Guide rev. 2.0*. Specification 2.0. Internet: `https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf` [Aug 09, 2022]. Object Managment Group, 2014.

[OMG15a] OMG. *OMG Data Distribution Service (DDS)*. Specification 1.4. Internet: `https://www.omg.org/spec/DDS/1.4/PDF` [Aug 09, 2022]. Object Management Group, Inc, 2015.

[OMG15b] OMG. *OMG Unified Modeling Language (UML)*. Specification 2.5. Internet: `http://www.omg.org/spec/UML/2.5/PDF/` [Aug 09, 2022]. Object Management Group, Inc, 2015.

[OMG16] OMG. *OMG Meta Object Facility (MOF) Core Specification*. Specification 2.5.1. Internet: `https://www.omg.org/spec/MOF/2.5.1/PDF` [Aug 09, 2022]. Object Management Group, 2016.

[OMG19] OMG. *UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)*. Specification 1.2. Internet: `https://www.omg.org/spec/MARTE/1.2/PDF` [Aug 09, 2022]. Object Managment Group, 2019.

[PHS22] Laurin Prenzel, Simon Hofmann, and Sebastian Steinhorst. *Rollback Sequences for Dynamic Reconfiguration of IEC 61499*. In: 2022 IEEE International Conference on Industrial Informatics (INDIN'22, in press). IEEE. 2022.

[Pla19] André Platzer. *The logical path to autonomous cyber-physical systems*. In: International Conference on Quantitative Evaluation of Systems. Springer. 2019, pp. 25–33.

[PP17] Laurin Prenzel and Julien Provost. *Dynamic software updating of IEC 61499 implementation using Erlang runtime system*. In: IFAC-PapersOnLine 50.1 (2017), pp. 12416–12421.

[Pre13] Tom Preston-Werner. *Semantic Versioning 2.0.0*. Internet: `https://semver.org/` [Aug 09, 2022]. 2013.

[Pri08] Paul J Prisaznuk. *ARINC 653 role in integrated modular avionics (IMA)*. In: 2008 IEEE/AIAA 27th Digital Avionics Systems Conference. IEEE. 2008, 1–E.

[PS21] Laurin Prenzel and Sebastian Steinhorst. *Automated Dependency Resolution for Dynamic Reconfiguration of IEC 61499*. In: 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE. 2021, pp. 1–8.

[PV14] Marco Panunzio and Tullio Vardanega. *A component-based process with separation of concerns for the development of embedded real-time software systems*. In: Journal of Systems and Software 96 (2014), pp. 105–121.

[Ram73] Chander Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets.* PhD thesis. Massachusetts Institute of technology, 1973.

[RH95] Mathias Rausch and H-M Hanisch. *Net condition/event systems with multiple condition outputs*. In: Proceedings 1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Automation. ETFA'95. Vol. 1. IEEE. 1995, pp. 592–600.

[RMF19] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. *The real-time linux kernel: A survey on preempt_rt*. In: ACM Computing Surveys (CSUR) 52.1 (2019), pp. 1–36.

[Rot06] Arnon Rotem-Gal-Oz. *Fallacies of distributed computing explained*. Internet: `https://www.se.rit.edu/~se442/doc/fallacies.pdf` [Aug 09, 2022]. 2006.

[SAS+09] Lui Sha, Abdullah Al-Nayeem, Mu Sun, Jose Meseguer, and Peter Csaba Olveczky. *PALS: Physically asynchronous logically synchronous systems*. In: Illinois Research and Tech Reports - Computer Science (2009). Internet: `https://www.ideals.illinois.edu/items/11944` [Aug 09, 2022].

[SCC+20] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. *Draco: Architectural and operating system support for system call security*. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE. 2020, pp. 42–57.

[Sch06] Douglas C. Schmidt. *Model-driven engineering*. In: IEEE Computer 39.2 (2006), p. 25.

[SDWB17] Philipp Schleiss, Christian Drabek, Gereon Weiss, and Bernhard Bauer. *Generic management of availability in fail-operational automotive systems*. In: International Conference on Computer Safety, Reliability, and Security. Springer. 2017, pp. 179–194.

[SH16] Gernot Starke and Peter Hruschka. *arc42*. Internet: `http://arc42.org` [Aug 09 2022]]. 2016.

[Sie18] Siemens AG. *SIMATIC IOT2000 Yocto board support package*. Internet: `https://github.com/siemens/meta-iot2000` [Aug 09, 2022]. 2018.

[SKKK03] Jamal H. Salim, Hormuzd M. Khosravi, Andi Kleen, and Alexey Kuznetsov. *RFC 3549 – Linux Netlink as an IP Services Protocol*. Specification. Internet: `https://www.rfc-editor.org/info/rfc3549` [Aug 09, 2022]. The Internet Society, Network Working Group, 2003.

[SRG96] Lui Sha, Ragunathan Rajkumar, and Michael Gagliardi. *Evolving dependable real-time systems*. In: IEEE Aerospace Applications Conf. Vol. 1. 1996, pp. 335–346.

[SS16] Tim Stock and Günther Seliger. *Opportunities of sustainable manufacturing in industry 4.0*. In: Procedia CIRP 40 (2016), pp. 536–541.

[SVZ13] Christoph Sünder, Valeriy Vyatkin, and Alois Zoitl. *Formal verification of downtimeless system evolution in embedded automation controllers*. In: ACM Transactions on Embedded Computing Systems (TECS) 12.1 (2013), pp. 1–17.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-oriented Programming*. Addison-Wesley, 1998.

[Tel15] Kilian Telschig. *Domänenspezifische Modelle zur Entwicklung generischer Automatisierungstechnik*. Master's Thesis. [Unpublished thesis]. University of Augsburg, 2015.

[Tha09] Sabu M Thampi. *Introduction to distributed systems*. In: arXiv preprint (2009). Internet: `https://arxiv.org/abs/0911.4395` [Aug 09, 2022].

[Tho22] Derek Thomas. *Evolution of edge control*. In: Control Engineering 69.1 (2022), pp. 17–20.

[TK17] Kilian Telschig and Alexander Knapp. *Towards safe dynamic updates of distributed embedded applications in factory automation*. In: 22nd IEEE Intl. Conf. Emerging Technologies and Factory Automation (ETFA). IEEE, 2017.

[TK19a] Kilian Telschig and Alexander Knapp. *Synchronous Reconfiguration of Distributed Embedded Applications During Operation*. In: 2019 IEEE International Conference on Software Architecture (ICSA). IEEE. 2019, pp. 121–130.

[TK19b] Kilian Telschig and Alexander Knapp. *Time-Critical State Transfer during Operation of Distributed Embedded Applications*. In: 2019 IEEE 17th International Conference on Industrial Informatics (INDIN). Vol. 1. IEEE. 2019, pp. 516–523.

[TKG16] Christos Tsigkanos, Timo Kehrer, and Carlo Ghezzi. *Architecting dynamic cyber-physical spaces*. In: Computing 98.10 (2016), pp. 1011–1040.

[TMM01] Lauren A Tewksbury, Louise E Moser, and Peter M Melliar-Smith. *Live upgrades of CORBA applications using object replication*. In: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001. IEEE. 2001, pp. 488–497.

[TSK18] Kilian Telschig, Andreas Schönberger, and Alexander Knapp. *A Real-Time Container Architecture for Dependable Distributed Embedded Applications*. In: 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE). IEEE, 2018, pp. 1367–1374.

[TSS+16] Kilian Telschig, Nikolai Schöffel, Klaus-Benedikt Schultis, Christoph Elsner, and Alexander Knapp. *SECO patterns: Architectural decision support in software ecosystems*. In: 2016 1st International Workshop on decision Making in Software ARCHitecture (MARCH). IEEE. 2016, pp. 38–44.

[VEBD07] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. *Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates*. In: IEEE Trans. Softw. Eng. 33.12 (Dec. 2007), pp. 856–868.

[VT17] Maarten Van Steen and Andrew S. Tanenbaum. *Distributed systems*. Third. Also published at `distributed-systems.net` [Aug 09, 2022]. CreateSpace Independent Publishing Platform, 2017.

[War20] Warner, Jim and Small, Craig and Cahalan, Albert. *Linux Programmer's Manual: top - display Linux processes*. Internet: `https://manpages.ubuntu.com/manpages/xenial/man1/top.1.html` [Aug 09, 2022]. The Ubuntu Manpage Repository, 2020.

[WCM+02] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. *Linux security module framework*. In: Ottawa Linux Symposium. Vol. 8032. Citeseer. 2002, pp. 6–16.

[WFP07] Murray Woodside, Greg Franks, and Dorina C Petriu. *The future of software performance engineering*. In: Future of Software Engineering (FOSE'07). IEEE. 2007, pp. 171–187.

[Wil13] H. Paul Williams. *Model building in mathematical programming*. John Wiley & Sons, 2013.

[Wil15] Ryan Williams. *PLCs vs. PACs vs. IPCs*. In: Control Engineering 62.11 (2015), pp. 34–36.

[WK17] Eric W. Biederman and Michael Kerrisk. *Linux Programmer's Manual: namespaces - overview of Linux namespaces*. Internet: `https://man7.org/linux/man-pages/man7/namespaces.7.html` [Aug 09, 2022]. The Linux Kernel Organization, 2017.

[Woo09] Michael Wooldridge. *An introduction to multiagent systems*. John wiley & sons, 2009.

[WRO09] Michael Wahler, Stefan Richter, and Manuel Oriol. *Dynamic software updates for real-time systems*. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades. ACM, 2009, pp. 1–6.

[XCG+21] Lei Xu, Lin Chen, Zhimin Gao, Hiram Moya, and Weidong Shi. *Reshaping the Landscape of the Future: Software-Defined Manufacturing*. In: Computer 54.7 (2021), pp. 27–36.

[YCB+13] Tatu Ylonen, Aaron Campbell, Bob Beck, Markus Friedl, Niels Provos, Theo de Raadt, and Dug Song. *Linux Programmer's Manual: sshd - OpenSSH SSH daemon*. Internet: `https://linux.die.net/man/8/sshd` [Dec 22, 2020]. The Linux Kernel Organization, 2013.

[ZHD05] Wei Zhang, Wolfgang A Halang, and Christian Dietrich. *Specification and Verification of Applications based on Function Blocks*. In: Component-Based Software Development for Embedded Systems. Springer, 2005.

[ZHLL18] Lin Zhao, Feng He, Ershuai Li, and Jun Lu. *Comparison of time sensitive networking (TSN) and TTEthernet*. In: 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC). IEEE. 2018, pp. 1–7.

[Zim80] Hubert Zimmermann. *OSI reference model-the ISO model of architecture for open systems interconnection*. In: IEEE Transactions on communications 28.4 (1980), pp. 425–432.

[ZJ97] Pamela Zave and Michael Jackson. *Four dark corners of requirements engineering*. In: ACM transactions on Software Engineering and Methodology (TOSEM) 6.1 (1997), pp. 1–30.

[ZLMV10] Alois Zoitl, Wilfried Lepuschitz, Munir Merdan, and Mathieu Vallée. *A real-time reconfiguration infrastructure for distributed embedded control systems*. In: Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on. IEEE. 2010, pp. 1–8.