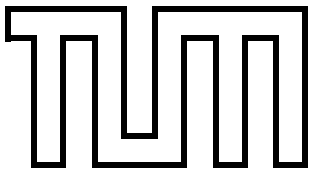


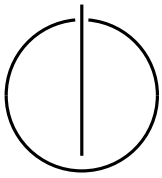
TECHNISCHE UNIVERSITÄT MÜNCHEN

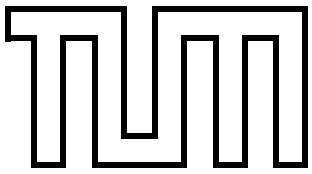
FAKULTÄT FÜR INFORMATIK

Master's Thesis in Robotics, Cognition and Intelligence

A dynamic, distributed and service-oriented robot control architecture

Sebastian Nagel





TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Master's Thesis in Robotics, Cognition and Intelligence

A dynamic, distributed and service-oriented robot control architecture

---

Eine dynamische, verteilte und service-orientierte Roboter Steuerungsarchitektur

Author:	Sebastian Nagel
Supervisor:	Univ.-Prof. Dr.-Ing. Darius Burschka
Advisor:	Dr.-Ing. Sami Haddadin
Submission Date:	March 17, 2014

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this masters thesis only supported by declared resources.

München, den 17. März 2014 .....

Sebastian Nagel

Sperrvermerk

Die vorliegende Masterarbeit ist für den Zeitraum von einem Jahr nach Einreichen streng vertraulich zu behandlen. Eine Veröffentlichung und Vervielfältigung der Masterarbeit ist - auch in Auszügen - nicht gestattet. Die Masterarbeit darf nur dem Erst- und Zweitgutachter sowie befugten Mitgliedern des Prüfungsausschusses zugänglich gemacht werden. Eine Einsichtnahme der Arbeit durch Unbefugte bedarf einer ausdrücklichen Genehmigung des Verfassers und des Unternehmens Kastanienbaum GmbH.

Blocking Notice

This master thesis is to be treated highly confidential in the period of one year from submission. Disclosure, publishing or duplication of the thesis - even in parts or in extracts - are not permitted without express authorization by the author and the company Kastanienbaum GmbH. This thesis is only hand over to correctors and members of the board of examiners.

Acknowledgements

I would like to express my special thanks to my advisor, Sami Haddadin for his insightful and visionary inputs in several discussions about the developed system and formalisms. Without him, this thesis would not have been possible.

Furthermore, I am very thankful to Professor Burschka for accepting me as a master student, asking the right questions when reviewing my work, but giving me the room to freely pursue this research topic.

I greatly appreciate the intense, but worthwile discussions with my colleagues Saskia Golz and Florian Meyer. In general, I am very thankful for the fun and inspiring startup atmosphere at Kastanienbaum with its former and current members.

Finally, I owe deepest gratitude to my friends and family, who supported me throughout my studies and this master thesis.

Abstract

Light-weight robotic manipulators get more affordable every year, and thus will be available to small enterprises and end-consumers sooner or later. This new audience demands for intuitive robot operating and programming environments focusing on robust and safe execution of robotic tasks. However, the currently available range of robotic software architectures struggle in bridging the gap between usability and functionality. The software framework and formalisms presented in this thesis build the foundation for proving task execution properties to ultimately support certifiability of the system. While exploiting the unique abstraction from real-time reflexes, safe and performant task execution is achieved by employing powerful high-level task modeling mechanisms and incorporating a novel formalization of component interaction. Structured operational semantics of the coordination and component model allow for clear statements about execution sequence and correctness of programmed robot tasks. Furthermore, the task modeling language in conjunction with the proposed system architecture proved very successful in various demonstrative scenarios. The results of this thesis represent an important first step to an intuitive, feature-rich, but reliable and safe framework for programming light-weight robots working in close co-operation with humans.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 System overview 3

1.3 Contributions and Outline 5

2 Background **7**

2.1 Robot architectures 7

2.2 Coordination languages 9

2.3 Model checking and formal semantics 12

2.4 Component-based robotics 13

2.5 Robot control and safety 13

3 RACE-Core Model **15**

3.1 RCM Syntax 16

3.1.1 Notation 17

3.1.2 Model elements 17

3.2 Intended semantics 26

3.2.1 Scope 26

3.2.2 Function language 27

3.2.3 Execution 28

3.2.4 Processing 29

3.3 Syntax extensions 30

3.3.1 Re-usability 30

3.3.2 Dynamic oracle . . . 31
3.4 Formal semantics . . . 34
3.4.1 Context and Environment . . . 36
3.4.2 Atomic layer . . . 37
3.4.3 Activation layer . . . 40
3.4.4 Micro Layer . . . 42
3.4.5 Quiescence . . . 46
3.4.6 Macro layer . . . 46
3.4.7 Execution . . . 47

4 RACE-Service Model 49
4.1 Service modalities . . . 50
4.2 RSM Syntax . . . 51
4.3 Formal semantics . . . 53
4.3.1 Semantic domain and communication . . . 53
4.3.2 Semantic rules . . . 55
4.4 Service interaction in RCM . . . 56
4.4.1 Revisited RCM syntax . . . 56
4.4.2 Revisited RCM semantics . . . 57

5 Model Validation 63
5.1 Expressiveness of syntax . . . 64
5.2 Intended and formal semantics . . . 71
5.3 Service interaction . . . 80

6 Conclusion 85

Bibliography 87

A Demonstrator 93

B Derivation trees 97

List of Figures

1.1 EMB.RACE system overview 3

2.1 Typical three-tier (3T) architecture 8

3.1 Relationships of RCM elements 18
3.2 *Grasp* state example 19
3.3 *Grasp* state with functions and actions 21
3.4 Composite state *Pick* with children *Move* and *Grasp* 22
3.5 Exemplary barrier chain and corresponding single barrier element 24
3.6 Complete example of a *Pick* task 25
3.7 Scopes in an RCM state-hierarchy 27
3.8 Re-use syntax extension 31
3.9 Dynamic oracle extension - extended syntax 33
3.10 Dynamic oracle extension - RCM syntax 33

4.1 Simplified robot service 53
4.2 Schematic visualization of modeled service interaction 56

5.1 *ObjectRecognition* state 65
5.2 Example implementation of *Pick* 67
5.3 *HapticGesture* state 68
5.4 *SearchObject* state 70
5.5 State with enabled port condition during action execution 71

5.6 Scenario which investigates port activation of composite state *Parent* when a connected child port gets activated 73

5.7 Exemplary state used for researching action execution sequence relative to transitioning semantics 76

5.8 Composite state with concurrently activating child ports (same condition) and identical destination 77

5.9 Composite state with simultaneously active child ports, transitioning to C and D, but preempting upon activation of C 78

5.10 Deadlock example, where the RCM and RSM instances get stuck 80

A.1 Demonstrator setup with LWR-III, gripper, display and the kinect depth camera 93

A.2 RACE-Pro client 95

A.3 RACE-One client 96

List of Listings

3.1 Function language - parameters, results and local variables 27
3.2 Function language - children 27
3.3 Function language - services 28
5.1 Function code of *GetAxis* action 69

Chapter 1

Introduction

1.1 Motivation

The focus of robotic system development is progressively shifting from big industry to small and medium-sized enterprises (SMEs) and end consumer applications [44]. Various manufacturers of robotic manipulators are pursuing this trend and new robots, capable of safely working together with humans, are presented on a regular basis. Most of these robotic systems are able to detect or even avoid unwanted contact with their environment. Prominent examples are the light-weight manipulators *LWR-III* [35, 4, 11] and *iiwa* [42] from KUKA, *Baxter* [56] developed by Rethink! Robotics or the *PR2* from Willow Garage [28]. With all these versatile systems getting more affordable and available to a broader public every year [57], the demand for intuitive software operating these robots is increasing as well.

Unfortunately, programming industry-grade robots is still a very challenging task, requiring quite some expertise in software-development and robotics. This leads to very high integration and programming costs, as well as long installation times. By reducing the set-up and reconfiguration times, higher efficiency and productivity on small batch sizes could be achieved, rendering robotic automation feasible for SMEs. While robotic middlewares [53, 17] have lowered the entry barriers for researchers and professionals by providing interchangable and re-usable software components, there are still several distributed computers controlling a single robotic task, prone to laborious (re-)configuration and long set-up times. To date, there does not exist a commonly accepted solution to intuitive and flexible programming of complex robotic systems.

However, the paradigm of "Plug-and-Produce" is widely desired to increase efficiency and lower integration time and cost. It is achieved by providing a self-explanatory and flexible system, which is espressive, modular and extensible. Among other factors,

simple and *reusable online*-programming are key-enablers to this paradigm for robotic systems. As these light-weight robots work in close vicinity and even in cooperation with humans, providing safe physical human robot interaction (pHRI) is of utmost importance. Interaction control [5] and safety measures [20, 29, 49] for light-weight robots promote the fence-less use of robots in direct coworking scenarios. Hence, the programming environment should also account for these modern techniques and provide *safety* on the program-level, as well as intuitive mechanisms for *error-handling*, and thus being robust to task failures. After all, software used for operating and programming robots is required to be *certifiable* in terms of correctness, security and safety. Consequently software components controlling a robot can be *validated* and *verified* against these properties via formal methods.

Robotic programming software partly fulfilling the outlined needs is currently only developed prototypical in academia (e.g. SMASH [12]) or for very simple robotic systems, where safety and certifiability is not an issue (e.g. UrbiScript and Gostai Suite [8]). Even though the programming paradigm of Rethink! Robotics is reasonably intuitive and aims to solve most of the stated issues, it is thightly coupled to their robot Baxter. The interface enables rapid development of "Pick 'n Place" tasks via *teaching by demonstration* on a very affordable platform and thus sets it into competition with traditional industrial robots, but cannot capitalize from highly sensitive collision detection, low-level collision reaction, precise assembly via joint-torque control and further safety features as realized with robots like the LWR-III. Similar issues exist with the programming model of Universal Robotics UR5 [48]. While the robot itself does not support advanced safety concepts of torque-controlled robot manipulators, the programming model already enables low set-up time and easy (re-)configuration. However, to be able to provide a straight-forward programming interface, the robot's functionality is reduced to a minimum and thus also restricts on possible application areas.

In conclusion, there is a significant lack of modular, certifiable, simple but flexible programming interfaces for modern high-tech robots, featuring advanced manipulation, safety and interaction modalities. This thesis introduces a system, which sets the foundation for a sophisticated programming and operation environment, satisfying the identified needs of industry as well as end-users, and elaborates on developed techniques necessary to realize this system called EMB.RACE.

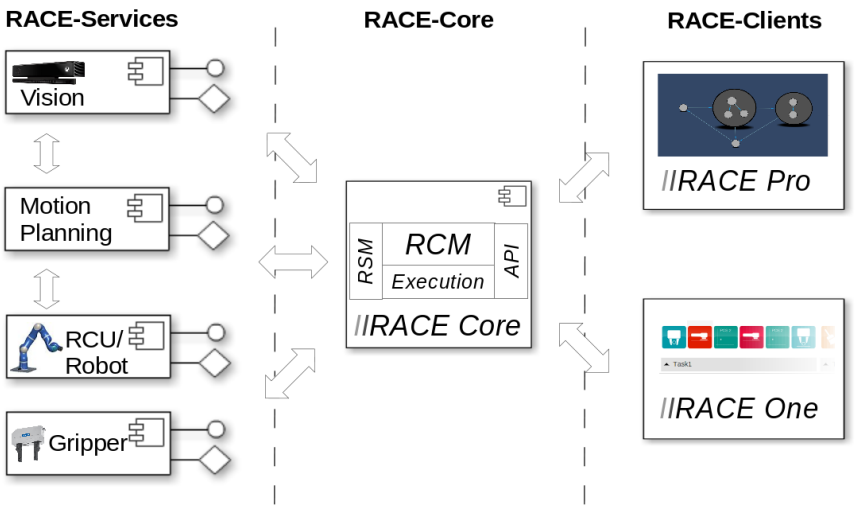


Figure 1.1: EMB.RACE system overview - separated into three parts: RACE-Services, RACE-Core and RACE-Clients, with exemplary vision, motion planning, gripper and robot services, as well as the two main clients RACE-Pro and RACE-One.

1.2 System overview

This section gives a coarse overview of the introduced EMB.RACE system, to better abstract this work's contributions. As already explained introductorily, a robot programming and operating environment has to fulfill a variety of requirements. EMB.RACE addresses these requirements by facilitating a dynamic, distribued and service-oriented approach to robot programming, enabled by uniquely abstracting robot control and low-level safety features [49]. The system architecture is schematically outlined in Figure 1.1, clearly visualizing the separation into the three parts of *RACE-Core*, *RACE-Clients* and *RACE-Services*, where the following paragraphs discuss the rationale for this separation.

Distributed system. The main use-cases of EMB.RACE are (re-)programming and operating robotic systems. In order to execute a program, user interfaces should not be required, but optionally available to visualize progress or change the program *online*. To support stand-alone operation, as well as dynamically connected client instances, a distributed approach is taken, where the RACE-Core component handles execution and can serve multiple RACE-Clients. The separation of core and clients also allows for different types of clients. Special user interfaces can be developed to target different audiences (e.g. a simplified client for novice users), whereas other systems can connect to core instances as client and process the execution model programatically (e.g. planning systems).

Robotic services. Tasks executed on a robotic system do not vary much in their need for computational capabilities (sensors, actuators, algorithms, etc.), but usually require a different configuration or logial sequence. Instead of re-implementing the logic for each task in slightly different variants or call sequences, a separation of concerns into coordination, computation, configuration and communication is typically made [54]. This abstraction often leads to component-based architectures like most of today's robot software frameworks (detailed in Section 2.4). In EMB.RACE, these components are called RACE-Services (computation), which can be dynamically accessed and parameterized by the RACE-Core (coordination). Typically, such components only advertise a generic interface, describing how they can be interacted with. RACE-Services go a step further and also publish their service state, including how it is affected by the advertised service functionality. To be able to communicate transparently over process and machine boundaries, either between RACE-Services or with the RACE-Core, a middleware solution is employed (communication).

Execution model. As the RACE-Core coordinates several RACE-Services to fulfill a certain robotic task, an appropriate formalism for modeling this coordination has to be defined. The execution model holds information about program sequence, parameterization and which services have to be queried. It has to be expressive enough to realize all required coordination methods (e.g. conditions or concurrency), but also easily understandable for both humans and artifical planners, as they interface the execution model. Especially, for human task developers, a graphical representation of the coordination model improves usability and enables visual programming. EMB.RACE utilizes a domain specific language (DSL) based on hierarchical finite state machine (HFSM) models like Harel statecharts [33], UML state diagrams [13] or Simulink Stateflow [1]. The newly developed formalism is called RACE-Core Model (RCM) and provides similar functionality as the mentioned HFSMs, but also captures service interaction. Along with clearly defined semantics, this incorporation of service communication benefits formal analysis of the robotic task and ultimately leads to provable functional correctness, which in turns increases general safety.

Re-usability and modularity. The separation of coordination, configuration and computation capabilities also promotes re-usability and modularity of robotic tasks, realizing faster switching between robot applications and task scenarios. Deploying tasks on an existing hardware setup gets as easy as downloading and starting an application on smartphones. By programming whole execution plans out of re-usable sub-tasks and skills, such robotic applications can be created and distributed, where the additional modeling of service interaction, enables EMB.RACE to create and execute tasks on heterogenous robotic systems.

1.3 Contributions and Outline

The contribution of this thesis is twofold: First, the RACE-Core Model (RCM) formalism for coordinating robotic systems is developed and presented as the execution model employed in RACE-Core instances of the EMB.RACE system. This formalization includes a mathematically defined syntax with graphical representation, intended and formal execution semantics of RCM. Although influenced by previous prototypes, the formalism was developed from the ground up in course of this thesis. Secondly, the generic RACE-Service Model (RSM) gets introduced in order to formalize interaction between the executed model of the RACE-Core instance and coordinated RACE-Services. While the development of these formalisms is comparable to related work in this field, formalization of interactions between the coordination model and its environment has not been proposed yet. The developed models are validated by analyzing expressiveness of syntax in various application scenarios. Furthermore, RCM and RSM semantics are validated by rigorously comparing intended semantics and formal derivation trees for constructed key scenarios. Although only included in the appendix, the proposed formalisms were prototypically implemented and used to realize various demonstrative robot applications.

The thesis continues with relevant background information in Chapter 2. The newly developed RACE-Core Model formalism is presented with its syntax, intended and formal semantics, as well as language extensions in Chapter 3. Afterwards, service interaction is introduced in Chapter 4 with the RACE-Service Model and necessary changes to RCM syntax and semantics. Both models are validated in Chapter 5, where parts of the formal derivation and demonstrator examples can be found in Appendix B. Finally, the conclusion is given in Chapter 6.

Chapter 2

Background

The work presented in this thesis discusses, relates or refers to several contributions in the topics of robot architectures in general, execution languages, formal analysis and component-based methodologies for robot software development. Also, it is based on previous research in robot control frameworks, safety and task coordination. The following sections will give an introduction to these related topics and constitute the background of this thesis.

2.1 Robot architectures

Software architectures empowering robotic systems have to solve unique challenges introduced by the heterogenous nature of these systems or the unstructured environment in which they operate. In order to fulfill certain tasks, robots need to establish communication between several different sensors and actuators, monitor execution of certain actions and handle unexpected situations. While staying reactive, a robot shall also accomplish a certain goal by deliberately executing appropriate sub-tasks.

Over time, a variety of different architectural styles have been developed, solving these special needs of robotic software. A thorough overview of important design aspects and the history of robotic system architectures is given in [41]. Considered as most relevant for this work are systems pursuing a layered architectural style as they were first introduced in [26] or [51]. A general version of the so-called three-tier (3T) architecture is depicted in Figure 2.1.

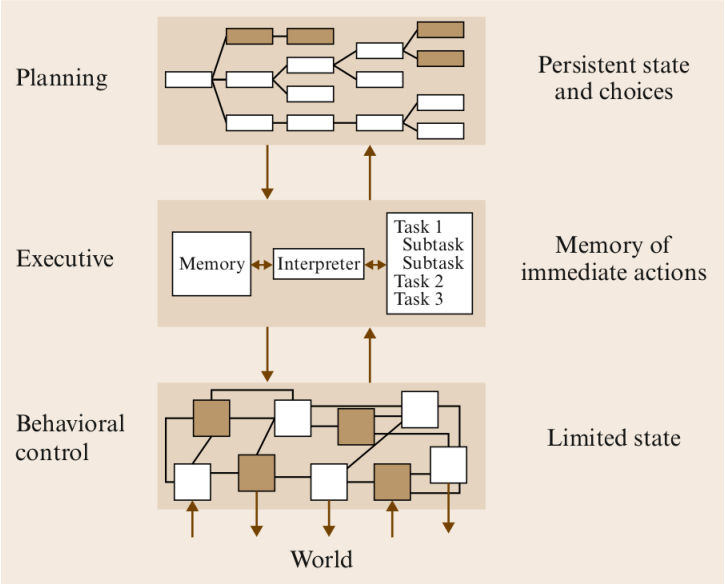


Figure 2.1: Typical three-tier (3T) architecture consisting of a behavior layer with basic and modular robot functionality; an executive layer, which coordinates low-level behaviors; a planning layer, deciding deliberately the robot's actions - from [60, p.191]

Layered architectures

One of the these 3T architectures is the LAAS architecture of autonomous systems (LAAS) [3]. It consists of three layers, namely the functional (behavioral) layer, an execution control layer and a decision (planning) layer, where the latter in turn consists of possibly mutiple layers, each acting in different temporal resolutions. Particularly interesting is the functional layer, which encapsulates robot action and perception algorithms into controllable communicating modules. The modules are created with the generator of modules (GenoM), which strongly relates to today's popular component-based robotic frameworks like Orocos [17] or ROS [53]. The most recently published GenoM3 [43] component model is also discussed in Section 2.4. The execution control layer is responsible for translating task requests from the decision layer into parameterized activations of functional modules and handles conflicts between modules by prioritization. Lastly, the decision layer consists of a planner and a supervisor, where the latter uses task plans of the former to send requests to the lower levels. This means, that the supervisor system named procedural reasoning system (PRS) takes the role of the executive as in other architectures. That is, decomposing and selecting tasks, monitoring their execution and reacting upon situations in a deliberate but time-bounded fashion (after lower levels may already have executed a reflexive action).

Another interesting architecture is the Coupled Layered Architecture for Robot Autonomy (CLARAty) [63] which was developed for operation of autonomous space rovers. CLARAty is a 2T architecture consisting of a functional and a decision layer. Also in this architecture, the functional level provides hierarchically decomposable abstractions via modules for motor control, navigation, manipulation etc., while the decision layer combines planning and execution capabilities for high-level decision-making. Although the decision layer was designed to support numerous planning and executive approaches, the commonly mentioned planner is Continuous Activity Scheduling Planning Execution and Replanning (CASPER) [19, 40] along with the Task Description Language (TDL) [61] and its corresponding executive. Together, they provide fast, reactive, but goal-driven handling based on resource and execution monitoring. There were efforts of integrating the Plan Execution Interchange Language (PLEXIL) with the accompanying Universal Executive (UE) [62]. Which is worth mentioning here, as PLEXIL was one of the main influences of RCM and gets introduced separately in the next section.

2.2 Coordination languages

The executive part of layered robot architectures is commonly responsible for translating high-level plans into activations of low-level behaviors, invoking them in proper sequence and parameterization, monitoring behavior execution and handling errors to some extent. To realize this reactive coordination, most executives utilize formalisms based on hierarchical finite state machines (HFSM) representing behavior sequence and keeping track of their execution. This section gives a short overview of prominent coordination languages.

Harel statecharts

Finite state machines (FSM) are well-suited for modeling reactive, event-based systems like robot coordination, but become unmanagable even for moderately complex systems. Due to necessary redundancy (e.g. identical reaction on the same event in several states) traditional FSMs are prone to the 'state-explosion problem', in which the complexity of the model tends to increase much faster then the complexity of the modeled system.

The *statecharts* formalism introduced by Harel [33] addresses exactly this issue by modeling similar behavior in hierarchical layers. Along with hierarchy, other mechansims like parallelism (via orthogonal regions - commonly named as AND states), inter-level transitions, history and local events, enable statecharts to model real-world reactive systems more faithfully than traditional FSMs.

A variety of other formalisms are based on Harel statecharts and there even exist different semantics for statecharts themselves [64]. This is mainly because the semantics of statecharts, as they were used in the 'original' STATEMATE environment [34], were published years after the language's syntax. However, a lot of research was conducted on statecharts and they are still very popular. The decision against statecharts as coordination language was made because some of its features impede compositionality and generally obstruct the definition of units with clear interfaces for re-use.

UML state diagrams

The Unified Modeling Language (UML) [13], is a general-purpose formalism originating from software-engineering, which allows to specify software-intensive systems in a variety of diagrams. These diagrams model software artifacts in their behavioral or static structure, can be used for code generation or even get interpreted and are executable.

The state diagram is the UML equivalent to HFSMs and is based on Harel statecharts. It features a similar syntax, although with some extensions, but provides slightly different semantics. One of the strenghts of UML state diagrams is its object-oriented focus, which is besides several semantic variation points (where implementations differ) the biggest drawback for utilizing UML state diagrams as coordination language in the robotic domain. The overall pursued 'lower-first' methodology (typical to classes specializing their superclasses) does not capture robotic task coordination very well. Nonetheless, due to its wide-spread acceptance, there are many efforts in formalizing the semantics of UML (refer to Section 2.3), which proved very useful for the work presented here.

Simulink Stateflow

Simulink is a block diagram environment for system-level design, simulation and code generation of embedded systems, where Stateflow is the graphical formalism for event-based modeling used in Simulink [1]. The Stateflow language is also based on Harel statecharts and provides near identical funtionality, but extends the syntax and semantics with condition actions, inner transitions, as well as a different event processing mechanism. Even though the high complexity, most of Stateflow semantics are formalized and thus designs (including generated code) can be verfied and validated.

Early prototypes of the system presented in this thesis were implemented using Stateflow, but the tight integration with Simulink complicates its use in robotic systems and the desire for a more dynamic and flexible coordination language eventually motivated the development of the RCM.

rFSM

The restricted Finite State Machine (rFSM) is a recently developed coordination model for robotic applications [39]. It adopts concepts from Harel statecharts and UML state diagrams, by critically reflecting both approaches, and focuses on coordination of robot functionality on a very low level, possibly running in hard real-time. The most significant divergence to both models is, that rFSM discards parallelism and advocates the modeling of concurrent functionality via distributed state machine instances. The reasons for this design decision hint on a different kind of robotic system involved in these scenarios. The approach of EMB.RACE allows to abstract low-level control and requires coordination on a higher level. After all, the semantics of rFSM lack a formal definition and are only described informally.

Urbiscript

The urbiscript language is an object-oriented scripting language tightly intregrated in the Urbi framework and is used for programming robotic systems using the Gostai Studio application [8]. It provides syntactic elements for concurrency and features a strongly event-based paradigm. Although urbiscript is not directly a hierarchical state machine, the development environment provides a hierarchical state machine representation which gets translated to urbiscript. The exact semantics of the state machine formalism are not publicly available, but seem to be similar to UML state diagram semantics.

PLEXIL

The Plan Execution Interchange Language (PLEXIL) is a coordination language developed by NASA aiming for a “compact, semantically clear and deterministic” execution model [25, 9]. Besides the aforementioned CLARATy project, PLEXIL is used in variety of systems, mostly concerned with autonomous rovers [27].

The language features a hierarchical composition of nodes, which are guarded by a set of conditions (start, end, repeat, invariant, pre and post). The latter can be interpreted as the equivalent of state transitions in HFSM models, but with a more comprehensive guarding mechanism using these conditions. PLEXIL provides also assignment and command node types to hold program logic for assigning values to variables of the defined node interfaces or call external commands (interacting with the robot) respectively.

Despite its interesting syntax and semantics, the formalism lacks a graphical representation and concepts like concurrency are not intuitive. However, the formal semantics of PLEXIL are well researched [22, 23] and served as a starting point for the formalization presented in this thesis.

2.3 Model checking and formal semantics

Verification and validation of hardware and software is crucial for safety critical systems like robots directly interacting with humans. While validation processes check that a system satisfies its requirements, verification is the evaluation whether a system works like it is intended to.

Model checking is concerned with the algorithmic verification of a system by exhaustively exploring the state-space of the system under consideration and checks whether a certain property holds true in all system states. Being a model-based verification technique, it requires a model describing the system behavior in a mathematically precise and unambiguous form. In fact, most inconsistencies, ambiguities or incomplete specifications are discovered already when modeling the system, prior to verification itself [7].

As the model to check will be a coordination language, besides a defined syntax, formal semantics are required to provide the foundation for checking language properties like determinism, compositionality, run-to-completion or termination, as well as enabling analysis of model instances. From *Concrete Semantics* by Nipkow and Klein [46]:

> Why formal semantics? Because there is no alternative when it comes to an unambiguous foundation of what is at the heart of computer science: programs. A formal semantics provides the much needed foundation for their work not just to programmers who want to reason about their programs but also to tool developers (e.g. compilers, refactoring tools and program analysers) and ultimately to language designers themselves.

There are three common semantic description methods, where these approaches are not competing, but are different techniques for different purposes, and possibly for different languages [45]:

- **Operational.** Describe the meaning of a program by individual steps and *how* syntactic elements are *executed*.
- **Denotational.** Semantics are modeled as mathematical structures and only the execution *effects* are considered.
- **Axiomatic.** The effect of executing language constructs are captured by *assertions* which may ignore certain aspects of the execution.

Semantics for coordination models

Formal semantics of coordination models (presented in the previous section) are typically defined using the *structured approach to operational semantics* (SOS) as it was introduced by Plotkin [52], which proved to be very flexible but intuitive [65, 32, 21, 22, 23]. Usually, SOS are defined on a labeled transition system (LTS) as the semantic domain, whose states are the closed terms over the model syntax and the (semantic) transitions of the LTS are inductively defined using rules of the form $\frac{premises}{conclusion}$ [2].

2.4 Component-based robotics

Most modern robot software architectures and frameworks pursue a component-based approach in modularizing computational capabilities to cope with the high system complexity and requirements like safety, reliability or fault tolerance [15]. Some examples of sophisticated component models are OROCOS/BRICS [17, 18], ORCA [14], GenoM3 [43] or SmartSOFT [59, 58]. All these frameworks face the challenges of engineering the software development process in robotics, with its high variability in algorithms, technologies and application scenarios. As already mentioned in Section 1.2, a separation of these challenges into four different design concerns is typically made, namely computation, configuration, communication and coordination [54, 16]. Techniques from model-driven engineering are used to guide the development process by encapsulating computational capabilities, providing them with appropriate means of (re-)configuration and establishing communication in-between. Also, component-based paradigms are strongly related to service-oriented concepts [55, 47], where the terms component and service are used interchangeably throughout this thesis.

2.5 Robot control and safety

The work presented in this thesis is partly based on previous research on safe robot control and physical Human-Robot Interaction (pHRI) [30], where EMB.RACE is intended to supersede the presented protoype for dynamic robot behavior programming. The next paragraphs will elaborate on some relevant approaches of this research field.

Light-weight robot design. Robots interacting with their environment have to cope with dynamic exchange of forces, which is achieved best by light structures featuring a low reflected mechanical impedance. On the other hand, a light-weight robot is more likely prone to intrinsic flexibility due to the components used, i.e. harmonic drive gears. This requires, additional joint torque sensing and accurate flexible joint dynamics modeling, which in turn, enable the detection of contact forces. The LWR-III robot

developed at the Deutsches Zentrum für Luft- und Raumfahrt (DLR) [4] is one of these light-weight, kinematically redundant, torque-controlled flexible joint robots, and got commercialized by KUKA [11].

Interaction control. One of the most common control schemes used for controlling direct physical interaction of flexible joint robots is impedance control [5]. The controller imposes a desired physical behavior on the robot and is very robust against inflicted external forces. For example, a robot can be controlled such that its end-effector behaves like a compliant cartesian mass-spring-damper system. Furthermore, collisions of the robot with its environment have to be detected by the control system in order to safely react to them [20]. As for collision reaction strategies, a variety of different control capabilities were developed, ranging from simple gravity compensation to more advanced strategies like trajectory scaling [29].

Behavior and reflexes. Besides the selectively mentioned control paradigms, the developed Robot Control Unit (RCU) also provides features like virtual walls, real-time motion planning or collision avoidance [49]. To switch between this vast amount of capabilities, a discrete action interface is used which defines an atomic robot action as a tuple $(command, behavior)$. While the command is nothing unusal (*move* or *stop*), the behavior part of a robot action is a very complex data structure describing the *operational* and *reflex behavior* of the robot. The operational behavior expresses the robot's particular motion and disturbance response during nominal task execution. Reflexes, on the other hand, specify a real-time reaction behavior on an activation signal (e.g. collision with certain severety). The Task Control Unit (TCU) then uses this interface to safely command the robot to execute task-relevant behavior.

Applications. The proposed safety features and the abstraction of robot control with RCU/TCU proved very successful in a variety of use-cases. Application scenarios range from co-operative bin-picking [31], interactive gesture-based tasks [24] to enabling robot teleoperation via a brain-machine-interface [36]. Although very diverse in their nature, these examples highlight the importance of pHRI in almost all scenarios of operating robots in a fence-less environment.

Chapter 3

RACE-Core Model

As the underlying formalism of the RACE-Core execution, the RACE-Core Model (RCM) captures coordination of all system components involved in a certain robotic task. Unlike most execution models in robot software architectures, the RCM is decoupled from the concrete robot, which still plays an essential role in most tasks, but is interfaced like any other service component (e.g. gripper or user-interface). This high-level abstraction is only possible by utilizing a discrete, complex, yet powerful interface to the low-level robot control framework [49]. Besides numerous robot control schemes and algorithms, this framework provides low-level safety reflexes, which relieve the burden of real-time coordination to provide safe robot operation. The focus of RCM as an execution model shifts therefore from robot- to task-level coordination.

While previous prototypes were strongly influenced by shortcomings of the originally employed Simulink Stateflow [1] model, the design process of the RCM was guided by the following requirements:

Intuitive. The syntax of RCM has to be easily processable by both, humans and machines, in order to support the planning process of the task. Hence, the model has to be mathemtically defined, unambigous, and readily supporting the definition of a formal semantics. Hierarchical finite state machines (HFSMs) are chosen as the basis for the RCM language, because they can be graphically represented, are formally clear and capture task structure very well. Moreover, the popularity of HFSM formalisms in several domains (e.g. software engineering or business process modeling) is beneficial.

Reactive. The main purpose of RCM, and coordination languages in general, is to execute certain program logic reactively on system events. These events can be caused by the nominal behavior of the coordinated system, but may also correspond to erroneous situations. Especially due to the high complexity of robotic systems, there are many points of failure in typical robotic tasks, upon which the execution model should be

able to react in time. Therefore, the modeling language should provide convenient and powerful mechanisms to describe reactive functionality of the executed task.

Deterministic. The behavior of model elements has to be clear and stay deterministic based on the syntacticly defined interface. This means, that no state should be hidden, and results in a functional approach with limited side-effects (in the model). All required parameters and produced results are explicitly defined for each model element, rather than implicitly accessing data in a some context. At first, the necessity of defining these interfaces or the limited access of functionality may seem laborious and impedes usability. But, this approach not only benefits a clear (compositional) semantics, but also forces task designers to properly abstract and separate functionality, which in turn improves re-usability.

Concurrent. In order to coordinate multiple aspects of a task using the same execution model simultaneously, some form of concurrency is required. The model needs to provide an intuitive and explicit method for splitting a task into parallel threads of execution and to synchronize this concurrent behavior.

Re-usable. The benefits of re-using already designed sub-tasks are obvious, but requires a modular approach with stable interfaces and appropriate means of parameterization. Furthermore, the re-use mechanism should be realized using an extension, which resolves into instances of the core language, as the re-use semantics should not inflict any changes to RCM execution semantics.

All these points did influence the design process of the RCM intensively and the next sections will elaborate on these while introducing RCM with its syntax (3.1), intended semantics (3.2), syntactic extensions (3.3) and formal operational semantics (3.4).

3.1 RCM Syntax

This section defines the syntactic classes of RCM mathematically and graphically. A form of set notation is used, which proved convenient for the semantics definition presented in Section 3.4. First, this notation is introduced hereafter, continuing with the formal definition of model elements and their graphical representation, along with discussions of made design decisions. The examplary scenario of robustly picking up an object is considered throughout this section to graphically illustrate the use of introduced model elements.

3.1.1 Notation

In the used notation, instances of syntactic classes are comprised by instances of other syntactic classes or defined sets. For example, let $\mathcal{A}$ and $\mathcal{B}$ be artifical sets with elements a and b. Then another syntactic class is defined as $\mathcal{X} = (\mathcal{A} \times \mathcal{B})$, specifying the syntax of elements x as tuples of the form (a, b). One could say that an element of $\mathcal{X}$ contains (or refers to) an instance of $\mathcal{A}$ and $\mathcal{B}$ each.

Element References

So far, the notation defines the syntactic structure, but in order to model all syntactic relationships of RCM, not only single instances have to be referred to, but also optional relations 0..1 and sequences with cardinality $1..n$ or $0..n$.

To achieve this, special sets are defined for all three representations. Given a set $\mathcal{A}$,

let $\mathcal{A}^0 = \mathcal{A} \cup \emptyset$ and let $\mathcal{A}^n$ be the set of all n-sized sequences over $\mathcal{A}$.

Then $\mathcal{A}^! = \bigcup_{i=1}^{n \in \mathbb{N}} \mathcal{A}^i$ and $\mathcal{A}^* = \bigcup_{i=0}^{n \in \mathbb{N}} \mathcal{A}^i$ can be defined.

Functions

Another convenient notation, which simplifies syntax and semantics definitions significantly is: given sets $\mathcal{A}, \mathcal{B}$ with instance $a \in \mathcal{A}$ and a function $f : \mathcal{A} \to \mathcal{B}$, the notation $f(a) \in \mathcal{B}$ or even shorter $a.f \in \mathcal{B}$ is used. Finally, functions may also be defined using this notation (besides other typical constructs, like the *set-builder notation*). For example: $a.f = \{b \mid \forall b \in \mathcal{B},\ b.g = \texttt{true}\}$ defines the function $f : \mathcal{A} \to \mathcal{B}$ to contain all elements of $\mathcal{B}$, which are mapped to `true` by another function $g : \mathcal{B} \to \{\texttt{true}, \texttt{false}\}$.

3.1.2 Model elements

The syntax of RCM consists of eight syntactic classes, namely: *State, Barrier, Port, Function, Condition, Action, Parameter* and *Result*. Figure 3.1 illustrates the relationships of these model elements using a (UML) class diagram related notation, where parameters and results are left out because their relations are straight-forward considering Definition 3.1. The auxiliary model class *Node* is defined as $\mathcal{N} = \mathcal{S} \cup \mathcal{B}$ to conveniently abstract common concepts of state and barrier, but is not considered as a syntactic class on its own. Finally, let $\mathcal{ID} = \mathcal{ID}_{\mathcal{S}} \cup \mathcal{ID}_{\mathcal{B}} \cup \mathcal{ID}_{svc}$ be the set of distinct state, barrier and service identifiers, $\mathcal{V}$ be a set of variable names and $\mathcal{C}$ be a set of valid (constant) values which might get assigned to variables, where type-safety is assumed. The boolean set is denoted as $\mathcal{I} = \{\texttt{true}, \texttt{false}\}$.

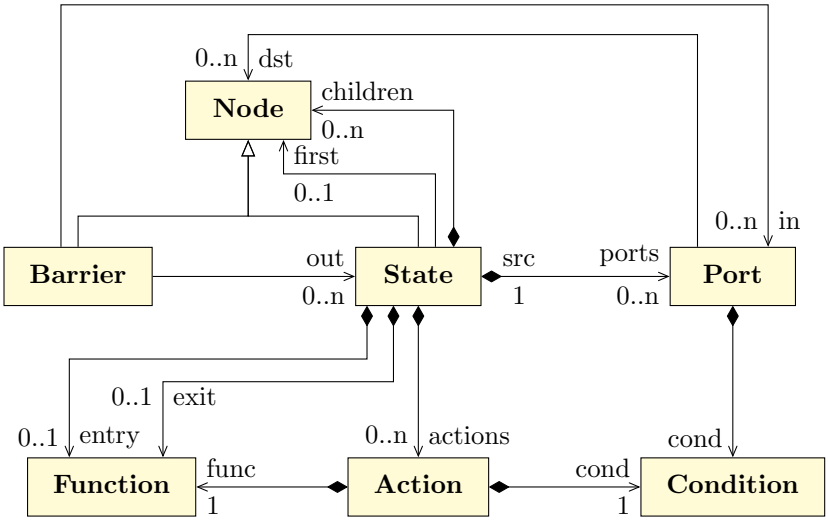


Figure 3.1: Relationships of RCM elements, parameters and results are left out for the sake of simplicity

States $s \in \mathcal{S} \subset \mathcal{N}$ are the central element of the RCM, encapsulating functionality and possibly consisting of other states:

Definition 3.1 (State)**.**

$$\mathcal{S} = (\mathcal{ID}_{\mathcal{S}} \times \mathcal{PA}^* \times \mathcal{RE}^* \times \mathcal{F}^0 \times \mathcal{F}^0 \times \mathcal{A}^* \times \mathcal{N}^* \times \mathcal{N}^0 \times \mathcal{P}^* \times \mathcal{I}) \tag{3.1}$$

Besides the required identifer $name(s) \in \mathcal{ID}_{\mathcal{S}}$, a state is comprised of $parameters(s) \in \mathcal{PA}^*$ and $results(s) \in \mathcal{RE}^*$, $entry(s) \in \mathcal{F}^0$ and $exit(s) \in \mathcal{F}^0$ functions, a set of $actions(s) \in \mathcal{A}^*$ and possible $children(s) \in \mathcal{N}^*$, of which, one is designated as $first(s) \in \mathcal{N}^0$. Finally a state has a set of $ports(s) \in \mathcal{P}^*$ and its activation state is described by the relation $active(s) \in \mathcal{I}$.

For convenience, additional functions are defined:

$$parent(s) = \{s_p \mid \exists s_p : \ s \in s_p.children\}$$

$$descendants(s) = \{c \cup c.descendants \mid \forall c \in s.children\}$$

$$siblings(s) = \{c \mid \forall c \in (s.parent.children \setminus s)\}$$

The graphical representation of a state is introduced in Figure 3.2, which shows an examplary state *Grasp*. The state model element is illustrated by a rectangle with rounded corners, where at least the name is shown in the first section. If the state instance holds further information like parameters, results or functions, they are listed

in the text section beneath the caption. The exemplary scenario of robustly picking an object, will be used throughout this section to introduce all syntactic elements.

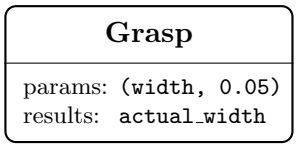


Figure 3.2: *Grasp* state example which takes a desired `width` as parameter and provides an `actual_width` as result value

Parameterization

Every state has an interface consisting of parameters $pa \in \mathcal{PA}$ and results $re \in \mathcal{RE}$, which syntacticly defines required and provided values respectively.

Definition 3.2 (Parameter)**.**

$$\mathcal{PA} = (\mathcal{V} \times (\mathcal{V} \cup \mathcal{C}) \times \mathcal{C}) \tag{3.2}$$

Parameters are described by a locally (in each state) unique $name(pa) \in \mathcal{V}$ and an expression $expr(pa) \in (\mathcal{V} \cup \mathcal{C})$ which is either a variable specifying another parameter/result as source or a constant (default) value. The resolved value of a parameter is denoted with property $value(pa) \in \mathcal{C}$.

Definition 3.3 (Result)**.**

$$\mathcal{RE} = (\mathcal{V} \times \mathcal{C}) \tag{3.3}$$

Results have a locally unique $name(re) \in \mathcal{V}$, and store a $value(re) \in \mathcal{C}$ during execution.

Parameters and results allow the explicit specification of a state's interface. The value of a parameter can either be constant ($pa.expr \in \mathcal{C}$) or derived from another parameter or result ($pa.expr \in \mathcal{V}$). The latter version would allow to use any arbitrary parameter as origin and thus, gets syntacticly *constrained* to parental parameters or results of siblings $\forall pa \in s.parameters$:

$$expr(pa) \in (\mathcal{V}_{parent} \cup \mathcal{V}_{siblings} \cup \mathcal{C})$$

with

$$\mathcal{V}_{parent} = \{pa.name \mid \forall pa \in s.parent.parameters\} \text{ and}$$

$$\mathcal{V}_{siblings} = \{re.name \mid \forall re \in sib.results, \forall sib \in s.siblings\}.$$

Functionality

Every state may specify some sort of functionality or behavior in terms of *entry* and *exit* functions, as well as a set of actions. Condition and function classes are defined using an *abstract function language*, as it is also done in other formalisms [32, 65]. This approach is common as it simplifies both syntax and semantics by only considering modeled effects of the function as a whole.

The main purpose of functions in RCM is to coordinate RACE-Services in their respective functionality, where these services provide a set of *operations*, as well as a variety of *events*, possibly holding received data. Although the definitions of services, conditions and functions will get refined when modeling service interaction in Chapter 4, a preliminary definition is required to introduce state functionality.

Definition 3.4 (Service)**.**

$$\mathcal{SVC} = (\mathcal{ID}_{svc} \times \mathcal{V}^* \times \mathcal{V}^*) \tag{3.4}$$

Each service $svc \in \mathcal{SVC}$ provides a unique service identifier $name(svc) \in \mathcal{ID}_{svc}$, a set of $operations(svc) \in \mathcal{V}^*$ and a set of $events(svc) \in \mathcal{V}^*$, where the names of operations and events are distinct. Operations may fail with an *error* or provide a *result*, while events hold *data* when received.

Definition 3.5 (Condition)**.** A boolean condition $bc \in \mathcal{BC}$ is defined as expression, written in a function language and evaluating to a boolean value $\mathcal{I} = \{\texttt{true}, \texttt{false}\}$.

Definition 3.6 (Function)**.** A function $f \in \mathcal{F}$ is a compound of several function language statements, where each statement may access and possibly change variable values.

Definition 3.7 (Action)**.**

$$\mathcal{A} = (\mathcal{BC} \times \mathcal{F}) \tag{3.5}$$

is the set of conditioned functions $a \in \mathcal{A}$, with $cond(a) \in \mathcal{BC}$ and $func(a) \in \mathcal{F}$ being the associated boolean condition and function of an action.

Even though functions and conditions do not (yet) capture whether they call any service operations or process event data, the functionality provided by services may already be used by an RCM element. Section 3.2.2 will introduce the function language as part of the intended semantics of RCM more clearly, but to legitimate the existence of conditions and functions as syntactic elements, some examples are listed hereafter and the exemplary *Grasp* state is extended with some demonstrative functionality in Figure 3.3.

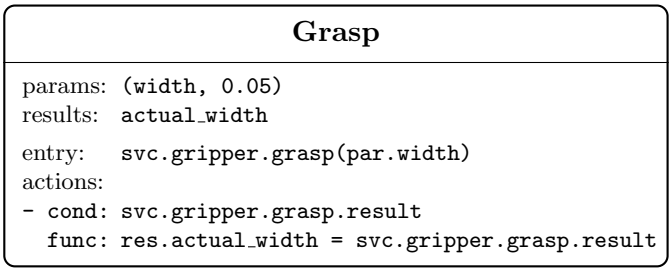


Figure 3.3: *Grasp* state with functions and actions, which call the `gripper` service operation `grasp` and write result `actual_width` on response (assuming the value is stored in the operation result)

Condition examples:

1. Accessing parameters:
   `par.value > 100`
2. Accessing results:
   `res.score == 0`
3. Checking availability of operation results (`grasp` of service `gripper`):
   `svc.gripper.grasp.result`

Function examples:

1. Processing parameters and results:
   `res.score = par.a + par.b`
2. Using local variables and accessing results (of child `Calc`):
   ```
   child_result = chi.Calc.res.value
   res.value = child_result * par.input
   ```
3. Calling operations of service components (`grasp` of service `gripper`):
   ```
   if(par.grasp == True):
       svc.gripper.grasp(par.width)
   ```

It is worth mentioning that these syntactic elements provide a fine-grained and powerful mechanism to react on certain events in the robotic system. Furthermore, a state *models deliberately a reactive set of actions*, which shall be executed by the system, as opposed to statechart-based formalisms, where executed actions may be specified (also) on transitions between system states (e.g. Stateflow [1]).

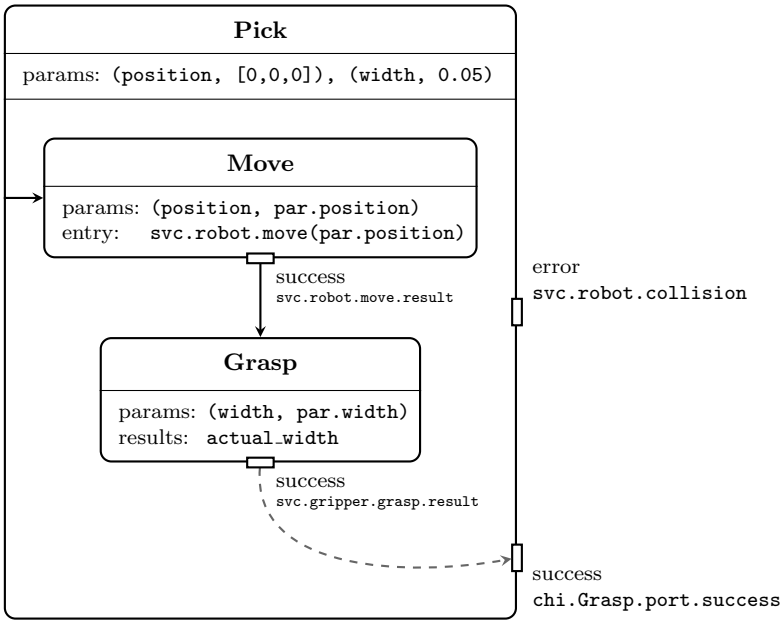


Figure 3.4: Composite state *Pick* holds children *Move* and *Grasp*, where *Move* is the *first* state of *Pick*. Both child states are parameterized by their parent and the *success* port condition depends on *Grasp's success* port

Composition

The state properties *children* and $first$ result in a hierarchy of states, in which each state may be composed of child states or barriers (introduced in Definition 3.9). Besides this *hierarchical composition*, states define with their ports also a *sequential composition*, where the latter is defined as:

Definition 3.8 (Port)**.**

$$\mathcal{P} = (\mathcal{V} \times \mathcal{S} \times \mathcal{N}^0 \times \mathcal{BC} \times \mathcal{I}) \tag{3.6}$$

Every port $p \in \mathcal{P}$ has a locally unique $name(p) \in \mathcal{V}$, a required source state $src(p) \in \mathcal{S}$, an optional destination node $dst(p) \in \mathcal{N}^0$ and a boolean condition $cond(p) \in \mathcal{BC}$. The activation status of a port is denoted in $active(p) \in \mathcal{I}$.

As specified by the property *src*, ports are associated with exactly one state. Thus, *ports* of a state s comprise an explicit interface of possible outcomes of this state. Each of these outcomes may represent a different situation, in which the state was left. The property *dst* then designates the next state, which gets activated once the port triggers on the condition specified in *cond*. The transition semantics are formally defined in Section 3.4.

Both, hierarchical and sequential composition are represented by the graphical model of a state and its ports, as seen in Figure 3.4. In case of a composite state, its children are placed in the area beneath defined parameters, results, functions and actions. Ports are illustrated as rectangles on the border of a state and the optional destination is connected by an arrow. Port conditions are written in `monospace` under the port name. The *first* state of a composition is denoted by a transition from the composite state itself to one of its children, where 'exiting' ports (port condition of parent port depending on child port activation) may be indicated graphically by a dashed transition line. Because of the sometimes limited space, details like port conditions, functions or actions may get omitted in the graphical representation, like it is the case for the exemplary *Grasp* state.

In order to define a consistent hierarchy, the following *constraints* have to be imposed on states and ports:

- $s \notin s.descendants$
- $s.first \in s.children$
- $c.parent = s, \forall c \in s.children$
- $p.src.parent = p.dst.parent$

have to hold $\forall s \in \mathcal{S}$ and $\forall p \in \mathcal{P}$.

The last constraint prohibits inter-level transitions, where this decision was made to improve compositionality. As outlined in [64], inter-level transitions (along with the history mechanism) impede the definition of compositional semantics. Furthermore, having strictly level transitions promotes syntactic interfaces provided by parameters, results and ports of a state. This, in turn, allows to implement re-usability mechanisms very easily as described in Section 3.3.

Concurrency

One of the main requirements of RCM, as stated in the beginning of this chapter, is the effective coordination of multiple distributed components in a robotic system. In order to fulfill this requirement, concurrent behavior has to be represented by the model. For this purpose another model element is defined:

Definition 3.9 (Barrier)**.**

$$\mathcal{B} = (\mathcal{ID}_{\mathcal{B}} \times \mathcal{P}^* \times \mathcal{S}^* \times \mathcal{P}^*) \tag{3.7}$$

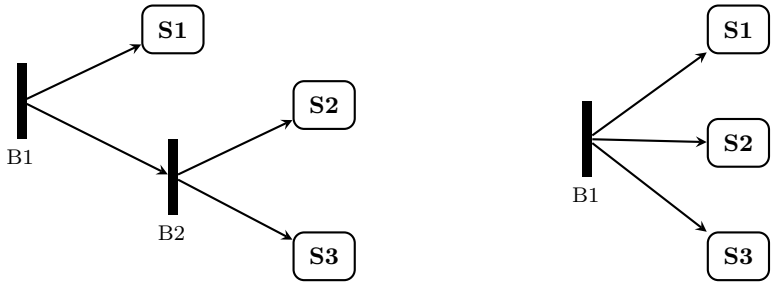


Figure 3.5: Exemplary barrier chain (left) and corresponding single barrier element (right)

Where a barrier $b \in \mathcal{B}$ specifies a $name(b) \in \mathcal{ID}_{\mathcal{B}}$, inbound ports $in(b) \in \mathcal{P}^*$ as well as outgoingly connected states $out(b) \in \mathcal{S}^*$. The activation state of a barrier is described by $active(b) \in \mathcal{I}$ and the set of already activated ports $activated(b) \in \mathcal{P}^*$.

Using properties *in* and *out*, barriers can be connected in between any port-state sequence. The intended functionality of a barrier is, that it activates all outbound states concurrently, after all inbound ports were activated (further discussed in Section 3.2).

As opposed to the *dst* property of ports, barriers may only specify transitions to states and not to other barrier instances. This decision was made because the semantics of a barrier chain would be roughly equivalent to a single barrier element, which connects all outgoing states itself. In fact, the only difference between these two alternatives (shown in Figure 3.5) exist in formal micro level semantics (Section 3.4) and are not straight-forward and possibly counter-intuitive.

An alternative way of modeling concurrency are *orthogonal regions*, as it is the case in Harel statecharts [33] or UML state diagrams [13], where the latter also defines barrier elements. The region approach integrates better with composite states, as they are simply extended with region syntax and semantics, but forcibly introduces additional hierarchical levels. States have to be composed in a separate hierarchy level in order to activate them concurrently and synchronization is only possible by leaving the composition.

By using the barrier model element, concurrency gets modeled explicitly and parallel threads in a state are obvious. Especially in the graphical model, concurrency can be seen at first glance (the model in Figure 3.6 uses a barrier element to enable the states *Haptic* and *Display* simultaneously). However, orthogonality of concurrent behavior (e.g. parallel threads of execution competing on state activation) is not enforced and it has to be checked (see Chapter 5) whether any constraints or even the developer's intention get violated.

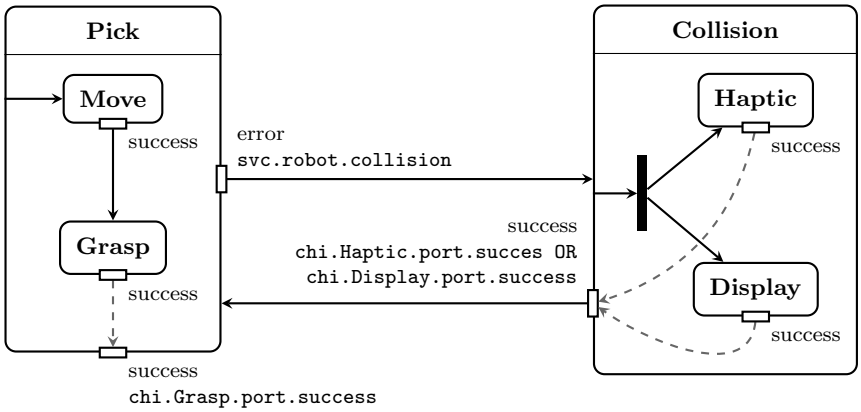


Figure 3.6: Complete example of a *Pick* task with handling of robot collisions, where details on parameterization and functionality are abstracted away for the sake of brevity

Summary

To summarize the introduced model elements, Figure 3.6 shows the complete *Pick* task, with its sub-states *Move* and *Grasp*, as well as an additional state *Collision*, which handles robot collisions as indicated by the port condition `svc.robot.collision`. As this example is based on a typical setup of EMB.RACE, where the robot reacts on collision using real-time reflexes [49], it is assumed that the safety relevant handling of the collision already occured when the coordination is in state *Collision*. The functionality of *Collision* consists of two alternative interaction schemes to re-enable the *Pick* task: haptic interaction with the robot or input via the user interface on a display. Both modalities are available at the same time as modeled by a barrier which branches into the two respective states *Haptic* and *Display*. When either one of these states exits via its *success* port, the state *Collision* also exits on port *success* as defined in the corresponding port condition.

This example also illustrates one of the most common problems in the robotics domain: Exception (collision) handling presented in the example could also be implemented on the level of state *Move*, or on a higher level, where multiple states like *Pick* are composed into a complex task. Designers of robotic applications have to decide whether such situations are handled either on *local* or *global* levels in the task hierarchy. This topic gets further discussed in Section 3.3 when introducing a mechanism which supports task developers in handling these cases on task-level.

3.2 Intended semantics

While the syntax of a language defines its static structure and how elements may be composed, semantics assign a meaning to these model elements and describe their behavior. Hence, the intended semantics describe intended and thus expected functionality of certain model elements. Often this intention is captured informally in prose or with diagrams, which do not suffice to unambiguously formulate model properties and rigorously check adherence to these properties (as opposed to formal semantics). However, an informal description of the intended meaning of language elements gives already a good understanding of said functionality.

As for RCM, this is introduced in four parts: First, the *scope* concept is defined and it is explained how functions and conditions written in a *function language* may access it. Next, the execution semantics of RCM are described by the taken steps when transitioning between states. Finally, topics concerning processing of events and service interaction are discussed.

3.2.1 Scope

The *scope* is a compound of internal and external information and is locally defined for each node in the state hierarchy and thus limits the *influence* a model element can have. Ports are provided with the scope of their source state. Figure 3.7 illustrates this by showing an example state hierachy with its associated scopes as triangles. With a given scope, all semantic values of the state belonging to it may get accessed, as well as activation information of the state's ports. Children of the scope state may be queried for their parameters, results and activation information, which includes whether a port is active or not.

Utilizing the scope, the behavior of following model elements can be defined:

- *Conditions* may *read* all aforementioned values of a scope to evaluate.
- *Functions*, on the other hand, can additionally *change* results and/or local variables when they are invoked.
- *Actions* are functions with a condition associated to a state and they are executed everytime when their condition evaluates to true while the state is *active*.
- Port conditions determine when the corresponding *Port* has to activate and trigger a transition to its destination node, also only when their source state is *active*.

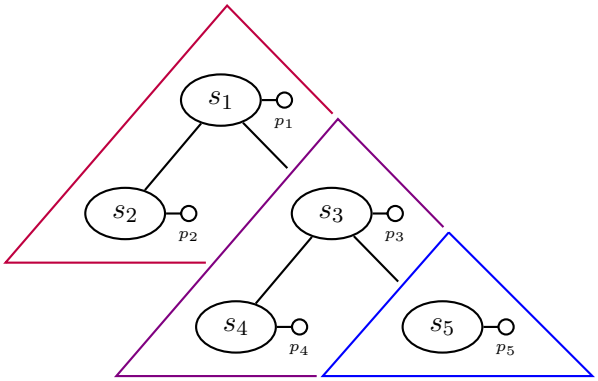


Figure 3.7: Scopes in an RCM state-hierarchy with associated ports, where the triangle shape indicates accessible model elements and thus represents a scope's *influence*. For example, the top-most (purple) scope belongs to s_1, to which it provides full access, whereas of s_2 and s_3 only parameters, results and (port) activation status are accessible.

3.2.2 Function language

As already stated in Section 3.1, the syntax of functions and conditions is defined with an *abstract function language*. This language is used to define the syntax of a function or condition and therefore has to be able to access scopes as well. Typically, an imperative language is chosen to describe in a sequence of statements, what effects the function should have. To illustrate the intent of this approach, some example function statements which access the scope of a state, are presented hereafter in the imperative language Python[1]:

Listing 3.1: Function language - parameters, results and local variables

```
par.value < 100
res.value = par.value * par.value
res.error = True
var.data = par.value
var.data == 'case1' or var.data == 'case2'
```

Listing 3.2: Function language - children

```
chi.Move.active == True
chi.Move.port.success
chi.Monitor.res.value > 100
```

[1]Python Language Reference, version 2.7 available at `http://www.python.org`

Listing 3.3: Function language - services

```
svc.gripper.grasp(0.05)
svc.gripper.grasp.result
svc[par.robot].move.error
svc[par.robot].state.collision == True
```

In the used instantiation of the function language, parameters, results, variables and services are accessed via the reserved identifiers `par`, `res`, `var` and `svc` respectively. Concrete instances in these collections can be queried using dot notation or via the array-access operator `[]`, where the latter is intuitive to use when for example the service to use is parameterized by an identifier as well (see Listing 3.3). Furthermore, any Python functionality could be imported, but allowed module and operations will be whitelisted to support the developer *not* to block the process and encourage asynchronous programming where possible.

3.2.3 Execution

Every RCM state is executable, where execution denotes the evaluation of a state and referred model elements as specified in the state's syntax. In order to execute a state it has to be specified in what sequence and how these model elements are evaluated. Even though the execution process is primarily defined by state transitions, describing the progress of a model, it also includes the coordinated invocation of specified actions, entry- and exit-functions.

The intended execution semantics can be described best by enumerating and assigning semantics to the model elements involved in the *sequential composition* of a state:

Port. While the source state of a port is active, it gets activated when the associated port condition evaluates to `true`. Once a port gets activated, the corresponding source state deactivates, the destination (either a state or barrier) gets activated and the port itself gets deactivated afterwards.

State. A state can be deactivated by its parent or one of its ports, which first leads to deactivation of all children (states and barriers), followed by invocation of the state's *exit* function. On activation, the *entry* function gets executed before the *first* child gets activated. Whenever a child state of a composition gets activated, the child's parameters have to be resolved first in the parent scope. Only when all parameters are provided with values, a state may get activated. While active, states evaluate their actions and execute them if the condition is fulfilled, before possible transitions of child ports or barriers are considered. However, port activations precede propagation to child states.

Barrier. Every barrier keeps track of inbound port activations via the property *activated*. When all connected ports have been activated, the barrier activates as well. This results in the simultaneous activation of all outgoing states and the barrier deactivates afterwards.

As already pointed out in some cases, the precedence of evaluation is top-down, which means, that conditions and thus transitions of upper layers in the state hierarchy are considered first and may *preempt* evaluation of lower levels. This precedence based on the hierarchical structure is often referred to as *structural priority* [34, 39] and is defined for most HFSM based formalisms because it decreases non-determinism caused by conflicting transitions (the semantics of UML even define lower-first priority). The example of Section 3.1, as shown in Figure 3.4, contains a typical configuration, when these semantics are necessary: Port *success* of composite state *Pick* depends and triggers on activation of a child port. The parental port has to be evaluated first and thus preempt any active children, before the child port deactivates, which could in turn result in a transition and activation of another state.

Evaluation order is an issue, which might not be clear from the informal description provided so far. The preemption mechanism via structural priority already solves most of the transition conflicts, where two or more ports would activate at the same time, i.e. the cases with different hierarchical levels. However, if conditions of two or more ports of the same state evaluate to `true` at the same time, the decision which port gets activated and transitions is prone to non-determinism. Usually, this issue is solved by requiring mutually exclusive guard conditions or by assigning priority numbers to possible conflicting elements, here ports.

Consequently, this issue can be generalized and also applies to multiple actions of a state which could have simultaneously fulfilled conditions, or the evaluation order of children, i.e. which child port gets transitioned first when multiple are active. The current approach of dealing with this problem is, to use inherent indices originating from collections used in the implementation. But as these indices are not intuitive to the task designer, explicitly assigned priorities might have to be introduced at some point. The mentioned scenario gets analyzed with more detail in Chapter 5 by validating intentional and formal semantics of RCM in such situations.

3.2.4 Processing

Practically all RCM instances interact with services by calling operations or processing operation results and service events. Service interaction is a core part of RCM as it is the equivalent to the event mechanism in traditional finite state machine formalisms (e.g. Harel statecharts) and thus essential for progressing the coordination model. Fur-

thermore, conditions correspond to guards and triggers of statecharts at the same time, enabling both, port activations and action execution in RCM. Hence, the model progresses only in response to received stimuli from services and does not transition if no such input is present.

The *synchrony hypothesis* was introduced by the developers of Esterel [10] and assumes, that a system reacts immediately on external events and creates its output instantaneously. This also includes the requirement, that any calculations, function calls, communication or data handling take no time. However, the synchrony hypothesis also states, that reactive tasks do not have to be exected in zero time, but it suffices to react *in time*, i.e. before the next system event occurs. This means, that events from the environment are processed faster than they occur. As a consequence, the environment remains invariant while the system processes an event.

According to [33, 34], statecharts comply with the synchrony hypothesis in an asynchronous time model where every transition takes zero time, while other formalisms describe similar semantics. Furthermore, *run-to-completion* semantics are defined for most coordination models, stating that processing of an event has to be completed first, in order to react on the next event.

As for RCM, received data from services gets only processed when the model has reached a complete state after handling all port activations and action executions. In between these so-called *macro steps*, the model gets evaluated against an invariant snapshot of the environment, where the local copy of the environment holds current operation results and received event data of all currently available services. All executed functions may not be long-blocking and asynchronous versions of data handling or communication are required to stay reactive and conform to the synchrony hypothesis.

3.3 Syntax extensions

The heavy use of explicit interfaces, like parameters and ports, results in a fully compositional syntax (and semantics) of RCM and supports the definition of syntactic extensions, also called syntactic sugar. These extensions may form a new langauge, which can be translated into native RCM syntax before execution.

3.3.1 Re-usability

In order to provide re-usable program components, RCM has to cope with re-utilization of certain states and even entire state hierarchies. The intended use of this mechanism is to provide a set of prebuilt modules in the programming environment, from which new

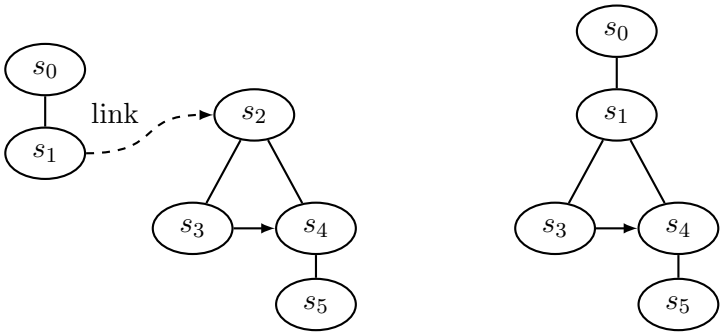


Figure 3.8: Re-use syntax extension: the linked state s_1 (left) gets expanded when translated into native RCM syntax (right)

tasks and modules can be composed. Re-usability is implemented using one of these aforementioned syntactic extensions, by introducing a link relation for states, which refer to another state (hierarchy) and expands to the referred elements when translated into native RCM. Formally, this is modeled by extending the syntax of states (Definition 3.1) with an additional relation $link : \mathcal{S} \rightarrow \mathcal{S}$. An example is shown in Figure 3.8. The advantage of this approach, is that RCM semantics do not have to cope with linked states, but the functionality can still be provided in a convenient and integrated syntax.

3.3.2 Dynamic oracle

The domain of physical human robot interaction is characterized by many system events upon which the task coordination has to react. The most classical example is that the robot is executing a certain task, when a human operator interferes by interacting with the robot. This interference is typically detected by the robot as a collision, which even may get classified as intended or uninteded. However, the task of the robot gets interrupted and task coordination has to deal with this unforseen and possibly erroneous situation.

To handle interaction on a task-level, as opposed to deciding locally on different layers of the currently executed state hierarchy, the task has to be able to get continued, aborted or skipped afterwards. The most common method is to re-enter the interrupted state, as achieved in various formalisms by some sort of *History* functionality. But as already pointed out, different policies might be desired on re-entrance of a composite state, and to achieve this, a mechanism called *Dynamic Oracle* gets introduced.

In general, said history mechanisms define state semantics based on the internal state by considering the last active child state as an entry point. This impedes the compositionality of state semanics by changing the combination of sub parts which constitute a

history-enabled state's semantics or by depending on the syntactic structure of history-enabled sub-parts [38, 64].

The property of compositionality could be achieved by exporting the history information to the state's interface. For example, the first state to enter would be part of the parameterization and gets changed during runtime. However, this would change the syntactic structure of said state and break any static analysis performed on a conrete RCM instance (checking model properties like satisfied parameterization).

The approach taken in RCM pursues the latter idea and provides the dynamic oracle extension, which allows to specify the first state to enter in *constant*, *parameterized* or even in an *externally* determined manner. This syntactic extension may get translated into native RCM syntax elements, and is realized by an expression determining the entry point. The different expressions for the respective dynamic oracle types are:

- *Constant*: refers to the child state's name directly or holds the name string itself, e.g. `chi.Child1.name` or `'Child1'`
- *Parameterized*: specifies a variable value which holds a name, e.g. stored in a parameter `par.first_state`
- *External*: determines the state to enter first in an external service component, possibly based on the current parameters, results or local variables of the state, e.g. `svc.Oracle.decide(par, res, var)`
- *History*: special expression to use the *History* strategy, as this requires additional memoization of the last active child state; denoted by a `history` keyword

An oracle-enabled state then gets transformed into basic RCM syntax by introduction of a *Dispatcher* state, which has ports connected to all entry candidates (other child states). The different cases are then enumerated in port conditions, which decide based on aforementioned expression the transition to take. The optional service call in the external variant would be processed in the dispatcher's entry and a corresponding action handling the response. Using this approach a variety of activation policies can be implemented. As an example, the common *History* mechanism is shown with oracle syntax in Figure 3.9 and the RCM translated model is shown in Figure 3.10.

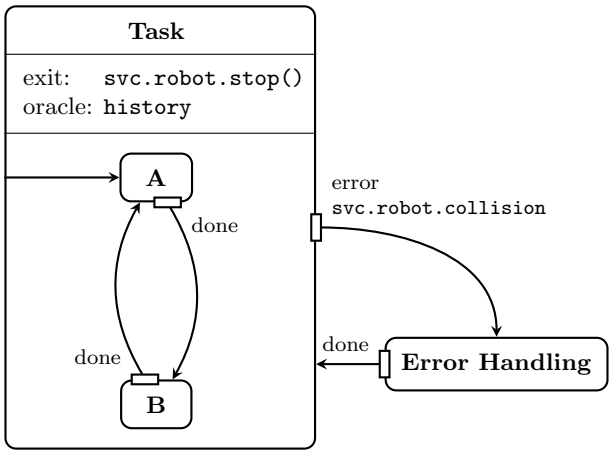


Figure 3.9: Dynamic oracle extension: A `history` type oracle state which continues *Task* execution when transitioning back from *Error Handling*. This syntactic extension needs to be translated to RCM syntax prior to execution.

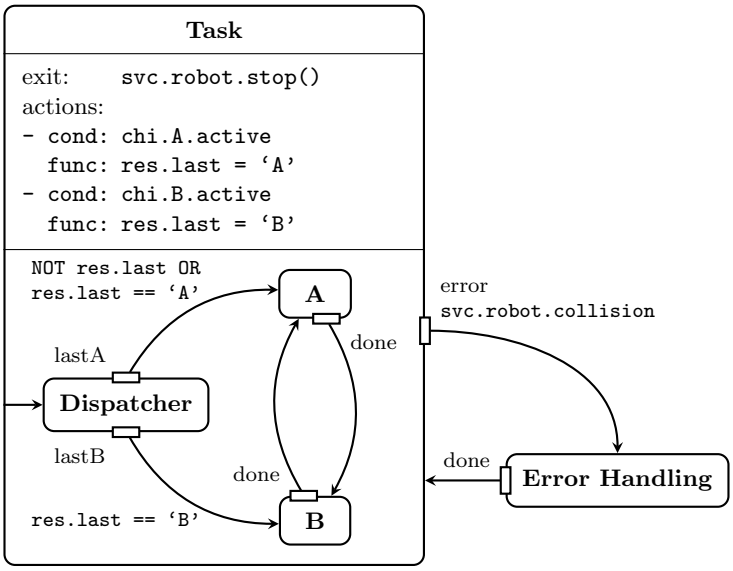


Figure 3.10: Dynamic oracle extension: Resulting RCM syntax after translating the `history` type oracle state.

3.4 Formal semantics

Robotic systems working in close vicinity with human operators, like the ones used with EMB.RACE, require high standards in regard to human safety while coordination gets very complex for increasingly sophisticated robotic tasks. These requirements create a need for advanced methods in analysis and verifcation, providing an opportunity for formal methods.

Unlike the informal description given in Section 3.2, formal semantics of RCM establish the foundation for proving language properties like determinism, compositionality, run-to-completion etc., while also enabling static analysis of RCM instances to check for violations of these properties. Furthermore, clear formal semantics serve as a reference for implementation of executives, compilers, parsers etc., while also supporting the language design itself. An overall iterative process is established, as new issues come up when formally specifying semantics after constructing the syntax.

The semantics of RCM are formalized using the structured operational semantics (SOS) approach of Plotkin [52], because it precisely captures the order of execution steps and thus gives an intuitive operational understanding while supporting the definition of a compositional and extendable semantics. Besides these features, the approach readily supports the translation to model checkers, which can be used to check the initially stated language properties, and was also used to formalize service interaction in Chapter 4.

Similar to the semantic framework of PLEXIL [21], the execution semantics of RCM is modularized using six layers of semantic relations:

- *Atomic*: contains basic evaluation and change of semantic values, condition evaluation, as well as function and action execution
- *Activation*: defines rules for activating and deactivating certain syntactic elements
- *Micro*: specifies transitional behavior of states and describes single steps of progress
- *Quiescence*: defines the run-to-completion of the micro relation
- *Macro*: describes how the model reacts on external events originating from services
- *Execution*: is the sequential evaluation of macro steps

As semantic domain, *labeled transition systems (LTS)* are chosen, to conveniently model service interaction via labels. For example, the semantics of a hypothetical semantic class $\mathcal{X}$ might be given by the labeled transition system $(\mathcal{X}, L, \rightarrow, x)$, where

- $\mathcal{X}$ is the set of system states,

- $L = (E \times A)$ is the set of labels,
- $\rightarrow \subseteq \mathcal{X} \times L \times \mathcal{X}$ is the transition relation, and
- x is the start state of the transition system.

This means, that each state of the transition system is represented by the current semantic state of a syntactic element $x \in \mathcal{X}$. In this exmaple, a label has the form $(\epsilon, \alpha) \in (E \times A)$, which can be interpreted in different ways, e.g. as the *input* and *output* of a transition as it was done in [65] with triggers and effects. The interpretation used in the semantics of this work will get introduced when utilized in Section 4.3. The transition relation $\rightarrow$ describes the semantic transition of states $x \in \mathcal{X}$ and is typically written $x \xrightarrow[\alpha]{\epsilon} x'$ instead of $(x, (\epsilon, \alpha), x') \in \rightarrow$. This notation can be read as: term x performs a semantic transition with input ϵ and output α to term x'. Note that x' indicates a change of the model element x, i.e. one of its semantic values gets altered, but a semantic transition does not necessarily result in a change of x. Finally, every transition system has to specify a start state, which is defined here as the initial state of the syntactic element.

The six layers of relations are initially defined using unlabeled transition systems, which are equivalent to labeled transition systems where the set of labels consists of only one element, i.e. $L = \{\tau\}$. As no semantics get assigned to this single element τ, it gets omitted. Furthermore, all semantic relations get inductively defined using SOS rules of the form $\frac{premises}{conclusion}$. Both, *premises* and *conclusions*, typically contain *contextual* semantic transitions of the corresponding LTS, where these contexts can be thought of as a parameterization, with which the semantic relation gets evaluated. For example,

$$C \vdash x \longrightarrow x', C'$$

describes the transition of x, in a context C, to the changed element x' and context C'. These changes sometimes get detailed using brackets, e.g. $x'[active = \texttt{true}]$ indicates that the semantic value *active* of x changed to `true`, or $C'[\Phi']$ implies a change of Φ. Consequently, the used SOS rules typically look like

$$\text{Rule1:} \frac{\begin{array}{c} C \vdash x.cond \rightsquigarrow \texttt{true} \\ C \vdash x \rightsquigarrow x'[active = \texttt{true}], C'[\Phi'] \end{array}}{C \vdash x \longrightarrow x', C'}$$

with the name of the rule on the left, premises above and the conclusion below the line. This example states that x evaluates in C via the relation $\longrightarrow$ to x', when $x.cond$ evaluates in C to `true` and x evaluates to x' and C' (both via the relation $\rightsquigarrow$).

Note that the sequence of premises defines the evaluation order in a semantic rule, whereas rule definition order determines precedence of their applicability. In case of the *premise* being a tautology, the rule is called an *axiom* and the premise is usually left out.

When SOS rules are applied to *derive* a semantic transition, a so-called *derivation tree* is obtained. The root of the tree is the derived conclusion, the leaves are axioms and intermediate nodes are instantiated SOS rules. To illustrate this, another example rule is defined, which depends on Rule1 as a premise:

$$\text{Rule2:}\frac{\begin{array}{c} C \vdash x.active \rightsquigarrow \texttt{false} \\ C \vdash x \longrightarrow x', C' \end{array}}{C \vdash x \Longrightarrow x', C'}$$

The derivation tree of applying Rule2 on state x is then:

$$\frac{C \vdash x.active \rightsquigarrow \texttt{false} \qquad \dfrac{C \vdash x.cond \rightsquigarrow \texttt{true} \quad C \vdash x \rightsquigarrow x'[active = \texttt{true}], C'[\Phi']}{C \vdash x \longrightarrow x', C'}}{C \vdash x \Longrightarrow x', C'}$$

3.4.1 Context and Environment

As stated before, the formal semantics of RCM are based on *contextual* semantic transitions. While contexts for these semantic relations can be chosen arbitrarily (even model elements), the used contexts are the environment Ω, an abstraction thereof Σ and the internal model state Δ.

Definition 3.10 (Environment)**.** The environment Ω is defined as the aggregation of services, with which an RCM instance interacts with. It holds the current state of service instances and provides appropriate means of communication between these processes. The latter gets utilized and defined in detail when formalizing service interaction.

Definition 3.11 (External Context)**.** The external context is denoted as Σ and captures a snapshot of an environment Ω. Thus, it provides the same functionality, but is not directly influenced by service interaction, as it stays invariant during *macro* steps on execution.

Definition 3.12 (Internal Context)**.** The internal context Δ holds the current state of the execution model and is used to describe the semantic state of contained model elements, where it is defined for every node in the model hierarchy.

The current state of model elements is described by four aspects contained in Δ:

- parameter values Π
- result values Ξ
- variable values Λ
- activation information Φ

Contexts will be used in several semantic rules, primarily to evaluate conditions or functions appropriately and to indicate how model elements change their current semantic state. Context changes are indicated by a prime Δ' and the change may be detailed in terms of which aspects are affected using $\Delta'[\Phi']$ for example to indicate activation changes. However, most changes are denoted directly on the corresponding state of the model element to better distinguish on what changed explicitly. For example, $\Delta \vdash s \rightsquigarrow s'[active = \texttt{true}], \Delta'$ states, that RCM state s gets activated in context Δ, and in turn results in the changed context Δ'. Hence, state s is *active* in context Δ', which may get evaluated by the atomic relation with $\Delta' \vdash s.active \rightsquigarrow \texttt{true}$.

Some semantic rules have multiple premises evaluating in a context and besides their denotation order, the context in which they evaluate also specifies the sequence of evaluation. In such cases a premise evaluating in contexts Σ and Δ may result in context Δ', where a second premise then specifies its dependency on context Δ' instead of Δ.

If semantic rules are defined with multiple contextual semantic transitions, the appropriate contexts (e.g. $\Delta_c, \forall c \in s.children$) sometimes have to get derived from another context, which is denoted by $\Delta \multimap \Delta_c$, and resulting contexts can propagate back via $\Delta'_c \rightharpoonup \Delta'$.

Corresponding to the intended scope semantics presented in Section 3.2.1, the context Δ is *scoped* for each syntactic element. This scoping strictly limits the influence of model elements to lower levels. Typically, only the interface of own children and ports is queried, as well as only direct sub-level contexts may get derived from Δ via $\multimap$, to benefit compositionality of the semantics.

3.4.2 Atomic layer

The first and most basic semantic layer defines elementary rules for evaluating and changing semantic values of model elements. Besides specifying when, how and with what effects these values change, the atomic layer also consists of rules for condition, function and action evaluation (execution). All other semantic layers build upon the rules of this layer.

Semantic values of RCM elements

These rules define the atomic change or evaluation of the current value of semantic properties like *active, activated* and *parameter values*. States, barriers and ports all define the *active* property for describing their current state of activation, where three rules specify activation, deactivation and evaluation of this *active* value $x \in (\mathcal{S} \cup \mathcal{B} \cup \mathcal{P})$:

$$\text{Active: } \frac{}{\Delta \vdash x.active \rightsquigarrow \{\texttt{true}, \texttt{false}\}} \tag{3.8}$$

$$\text{Act: } \frac{\Delta \vdash x.active \rightsquigarrow \texttt{false}}{\Delta \vdash x \rightsquigarrow x'[active = \texttt{true}], \Delta'[\Phi']} \tag{3.9}$$

$$\text{Deact: } \frac{\Delta \vdash x.active \rightsquigarrow \texttt{true}}{\Delta \vdash x \rightsquigarrow x'[active = \texttt{false}], \Delta'[\Phi']} \tag{3.10}$$

In addition to being active themselves, barriers also keep track of activations of connected inbound ports via the property *activated*, which is defined by the following two rules with $b \in \mathcal{B}$ and $p \in \mathcal{P}$, where $\overset{\cup}{=}$ is defined as the union assignment operator, i.e. $A = A \cup B \iff A \overset{\cup}{=} B$.

$$\text{BIn: } \frac{p \in b.in}{\Delta, p \vdash b \rightsquigarrow b'[activated \overset{\cup}{=} p], \Delta'[\Phi']} \tag{3.11}$$

The *activated* status of a given port $p \in \mathcal{P}$ in a certain barrier $b \in \mathcal{B}$ is determined by:

$$\text{Activated: } \frac{}{\Delta, p \vdash b.activated(p) \rightsquigarrow \{\texttt{true}, \texttt{false}\}} \tag{3.12}$$

Parameters, results and local variables, each possess a current semantic *value*, but as the values themselves are captured by the context Δ, no rules need to be defined for changing and reading these values. However, the expression of parameters $pa \in \mathcal{PA}$ has to be resolvable to a value $c \in \mathcal{C}$ when entering a state.

$$\text{Resolve: } \frac{}{\Delta \vdash pa.expr \rightsquigarrow c \in \mathcal{C}} \tag{3.13}$$

Function language evaluation

RCM conditions, functions and actions are defined using the distinct *function language*. The formalization of execution semantics however does not require a detailed description of individual function statements and thus gets abstracted to function evaluation as a whole in rule definitions. This approach is common when formalizing coordination

languages (e.g. Stateflow [32]) and simplifies semantics of elements like functions. However, in order to formalize service interaction, it is necessary to describe the behavior of individual function statements in greater detail. For this purpose, the following semantic rules get refined in Section 4.4.

Conditions

Boolean conditions $bc \in \mathcal{BC}$ get evaluated on contexts Σ and Δ to a boolean value, and access these contexts using function language expressions as described in Section 3.2.2.

$$\text{Cond: } \frac{}{\Sigma, \Delta \vdash bc \rightsquigarrow b \in \{\texttt{true}, \texttt{false}\}} \tag{3.14}$$

Functions

Similar to conditions, functions $f \in \mathcal{F}$ access contexts Σ and Δ on evaluation, but may additionally change current *result values* Ξ and *local variables* Λ of the given internal context Δ.

$$\text{Func: } \frac{}{\Sigma, \Delta \vdash f \rightsquigarrow f, \Delta'[\Xi', \Lambda']} \tag{3.15}$$

Actions

As actions $a \in \mathcal{A}$ are conditioned functions, the semantics are based on condition and function evaluation, where the function is only executed if the condition evaluates to `true`. Moreover, any action may only execute its function once every macro step, which is achieved by setting the *executed* flag in context Σ.

$$\text{Action1: } \frac{\begin{array}{l}\Sigma \vdash executed(\Sigma, a) \rightsquigarrow \texttt{false} \\ \Sigma, \Delta \vdash a.cond \rightsquigarrow \texttt{true} \\ \Sigma, \Delta \vdash a.func \rightsquigarrow a.func, \Delta'[\Xi', \Lambda']\end{array}}{\Sigma, \Delta \vdash a \rightsquigarrow a, \Delta', \Sigma'[executed \overset{\cup}{=} a]} \tag{3.16}$$

For the sake of breviety, the detailed effects of semantic transitions are sometimes left out, as it is the case for the changed context Δ' in this rule. Also, as these three rules are the only ones considering *executed*, the change in external context Σ can be neglected in other rules.

$$\text{Action2: } \frac{\Sigma \vdash executed(\Sigma, a) \rightsquigarrow \texttt{true}}{\Sigma \vdash a \rightsquigarrow a} \tag{3.17}$$

$$\text{Action3: } \frac{\Sigma, \Delta \vdash a.cond \rightsquigarrow \texttt{false}}{\Sigma, \Delta \vdash a \rightsquigarrow a} \tag{3.18}$$

3.4.3 Activation layer

This layer provides auxiliary activation $\longrightarrow_{\uparrow}$ and deactivation $\longrightarrow_{\downarrow}$ relations to model activation and deactivation processes of certain model elements respectively. The rules in this layer are parameterized with the evaluation relation $\rightsquigarrow$ of the atomic layer. Modeling this layer of abstraction is convenient because upper layers can focus on defining transition semantics, while effects of activation and deactivation are captured by relations of this semantic layer.

Deactivation of states and barriers

First, deactivation of an already inactive state $s \in \mathcal{S}$ is modeled by *sequentially* deactivating all ports as indicated by the notation used in the second premise of this rule. Here, every port has to deactivate one after another, denoted by context indices on which the premise is evaluated. The initial context Δ^0 is derived by the given context Δ, where the $\rightharpoonup \Delta'$ notation is used instead of explicitly stating $\Delta^n \rightharpoonup \Delta'$ as a fourth premise line.

$$\text{SDeact1:}\ \frac{\begin{array}{c}\Delta \vdash s.active \rightsquigarrow \texttt{false} \\ \forall p_i \in s.ports:\ i = 1...n,\ \Delta \rightharpoonup \Delta^0 \\ \Delta^{i-1} \vdash p_i \rightsquigarrow p_i, \Delta^i\ \rightharpoonup \Delta'\end{array}}{\Sigma, \Delta \vdash s \longrightarrow_{\downarrow} s, \Delta'} \tag{3.19}$$

The second rule describes deactivation of a previously active state $s \in \mathcal{S}$, where first, deactivation is propagated to all children $c \in s.children$, before the state's *exit* function is executed and it gets deactivated. Propagating deactivation is done *synchronously* by deriving the appropriate contexts Δ_{c_i} from the current context Δ and evaluating the premise relation independently for all children. In the style of previous notation, this could be written:

$$\left.\begin{array}{l}\forall c_i \in s.children:\ i = 1...n,\ \Delta \rightharpoonup (\Delta_{c_1}...\Delta_{c_n}) \\ \quad \Sigma, \Delta_{c_1} \vdash c_1 \longrightarrow_{\downarrow} c_1', \Delta_{c_1}' \\ \quad ... \\ \quad \Sigma, \Delta_{c_n} \vdash c_n \longrightarrow_{\downarrow} c_n', \Delta_{c_n}'\end{array}\right\} \rightharpoonup \Delta'$$

This clearly shows that each premise is evaluated in independent contexts Δ_{c_i} and result in an altered contexts Δ_{c_i}', which cumulatively propagate 'up' to a changed context Δ'. However, the notation gets slightly simplified in the following rule, exploiting evaluation independence, but still illustrating applicability to all children and the cumulative result in context Δ'. In summary, it describes state deactivation and *preemption* of potential child states.

$$\text{SDeact2:}\ \frac{\begin{array}{l}\Delta \vdash s.active \rightsquigarrow \texttt{true} \\ \forall c \in s.children:\ \Delta \multimap \Delta_c \\ \quad \Sigma, \Delta_c \vdash c \longrightarrow_{\downarrow} c', \Delta_c' \ \multimap \Delta' \\ \Sigma, \Delta' \vdash s.exit \rightsquigarrow s.exit, \Delta'' \\ \Sigma, \Delta'' \vdash s \rightsquigarrow s'[active = \texttt{false}], \Delta'''\end{array}}{\Sigma, \Delta \vdash s \longrightarrow_{\downarrow} s', \Delta'''} \tag{3.20}$$

A barrier $b \in \mathcal{B}$, on the other hand, gets deactivated by simply clearing its activation information as described by the rule:

$$\text{BDeact:}\ \frac{\begin{array}{c}\Delta \vdash b \rightsquigarrow b'[active = \texttt{false}], \Delta' \\ \Delta' \vdash b' \rightsquigarrow b''[activated = \emptyset], \Delta''\end{array}}{\Delta \vdash b \longrightarrow_{\downarrow} b'', \Delta''} \tag{3.21}$$

Activation of states and barriers

Both, states and barriers, can be referred to as the $first$ element of a composite state or as the *dst* of a port. These are the only cases when the execution semantics result in an activation of a state or barrier. Hence, all activation rules are provided with a port p as context to distinguish these cases and optionally keep track of port activations.

State activation is modeled by setting the state *active*, executing the state's *entry* function and finally propagating activation to the $first$ child. Note that the $\forall$ quantifier is used to *optionally* apply the last premise, while the $first$-relation has cardinality 0..1.

$$\text{SAct1:}\ \frac{\begin{array}{l}\Delta \vdash s.active \rightsquigarrow \texttt{false} \\ \Delta \vdash s \rightsquigarrow s'[active = \texttt{true}], \Delta' \\ \Sigma, \Delta' \vdash s.entry \rightsquigarrow s.entry, \Delta'' \\ \forall f \in s.first:\ \Delta'' \multimap \Delta_f \\ \quad \Sigma, \Delta_f, \emptyset \vdash f \longrightarrow_{\uparrow} f', \Delta_f' \ \multimap \Delta'''\end{array}}{\Sigma, \Delta, p \vdash s \longrightarrow_{\uparrow} s', \Delta'''} \tag{3.22}$$

Even though the port context does not affect any of the premises for activating a state, the empty set has to be passed as port context when propagating, because barrier semantics have to determine that the element gets activated as the first child in a composition. Additionally, a rule is defined for already active states, where the semantic transition can be applied without further evaluation and context Δ stays unchanged.

$$\text{SAct2:}\ \frac{\Delta \vdash s.active \rightsquigarrow \texttt{true}}{\Sigma, \Delta, p \vdash s \longrightarrow_{\uparrow} s} \tag{3.23}$$

Barrier activation (as first) In this first case, a barrier is the $first$ child of a composite state on activation. This is the only scenario where a $\emptyset$ port context is supplied, which is checked by the second premise, and results in a direct activation of the barrier element.

$$\text{BAct1:}\ \frac{\begin{array}{l}\Delta \vdash b.active \rightsquigarrow \texttt{false} \\ p = \emptyset \\ \Delta \vdash b \rightsquigarrow b'[active = \texttt{true}], \Delta'\end{array}}{\Sigma, \Delta, p \vdash b \longrightarrow_{\uparrow} b', \Delta'} \tag{3.24}$$

Barrier activation (by port). The second rule models the case, where a barrier gets activated by a port-issued transition and the corresponding port is supplied as context p. Using this context, barrier semantics register the port as *activated*. Once all ports were triggered the barrier element changes its semantic state to active. The third premise utilizes the $\forall$ quantifier to denote required applicability to all connected ports.

$$\text{BAct2:}\ \frac{\begin{array}{l}\Delta \vdash b.active \rightsquigarrow \texttt{false} \\ \Delta \vdash b \rightsquigarrow b'[activated \overset{\cup}{=} p], \Delta' \\ \forall p_i \in b.in : \\ \quad \Delta' \vdash b.activated(p_i) \rightsquigarrow \texttt{true} \\ \Delta' \vdash b \rightsquigarrow b'[active = \texttt{true}], \Delta''\end{array}}{\Sigma, \Delta, p \vdash b \longrightarrow_{\uparrow} b', \Delta''} \tag{3.25}$$

The last case, where not all ports got *activated* yet, is described by the following rule. The third premise has to hold for at least one *in*-port as indicated by the $\exists$ quantifier and the barrier remains inactive.

$$\text{BAct3:}\ \frac{\begin{array}{l}\Delta \vdash b.active \rightsquigarrow \texttt{false} \\ \Delta \vdash b \rightsquigarrow b'[activated \overset{\cup}{=} p], \Delta' \\ \exists p_i \in b.in : \\ \quad \Delta' \vdash b.activated(p_i) \rightsquigarrow \texttt{false}\end{array}}{\Sigma, \Delta, p \vdash b \longrightarrow_{\uparrow} b', \Delta'} \tag{3.26}$$

3.4.4 Micro Layer

Utilizing the *activation layer*, the micro layer semantics define the individual steps, which drive the execution. There are two kinds of these so-called *micro steps*: First, semantic transitions describing a model's *progress*, typically representing syntactic transitions, e.g. from active ports to destination states. The second type are *stuttering* transitions, which do not change currently active model elements, but may result in

changes of semantic values, e.g. execution of state actions reacting on an event. All micro layer rules are formulated in terms of RCM states and inductively define the $\longrightarrow$-relation in a corresponding LTS. Finally, another relation $\Longrightarrow$ is introduced to describe progression/stuttering of the whole model in its current semantic state Δ.

Progress

Preemption. Model progress always originates from port activations. Rule PTrigger is the first and thus highest prioritized micro layer rule. It describes the situation, where a port condition evaluates to `true` and deactivates the corresponding source state $s \in \mathcal{S}$, including preemption of potentially active child states as defined by rule SDeact2.

$$\text{PTrigger:}\ \frac{\begin{array}{l}\Delta \vdash s.active \rightsquigarrow \texttt{true} \\ \exists p \in s.ports : \\ \quad \Sigma, \Delta \vdash p.cond \rightsquigarrow \texttt{true} \\ \quad \Sigma, \Delta \vdash s \longrightarrow_{\downarrow} s', \Delta' \\ \quad \Delta' \vdash p \rightsquigarrow p'[active = \texttt{true}], \Delta''\end{array}}{\Sigma, \Delta \vdash s \longrightarrow s', \Delta''} \tag{3.27}$$

As the evaluation order of ports is not explicitly defined, this may lead to an unintentional prioritization of certain ports when multiple conditions evaluate to `true` in the same micro step. Although being deterministic by selecting the first (i.e. with lowest index) port, whose condition is fulfilled, these semantics are not explicit for the task designer. In general, these order-dependant semantics (also in other rules) may get improved in future versions by introducing some sort of priority syntax. In the case of 'conflicting' port conditions, the best-practice of designing *mutually exclusive* port conditions resolves this ambiguity as well.

Port transition. Once a port is active, a syntactic transition to its destination state is taken, as described by the next rule - PTrans1. It defines the individual semantic transitions comprising a port-issued syntactic transition. Let the function $active\_ports : (\Delta \times \mathcal{S}) \rightarrow (\mathcal{P} \times \mathcal{N})$ list all currently (in context Δ) active child ports $p \in \mathcal{P}$ of state $s \in \mathcal{S}$, with their destination $d \in \mathcal{N}$, i.e. all ports which are ready for transitioning. In order to activate a syntactically defined destination d, all its parameters have to be resolved in the activating context Δ, leading to updated parameter values of d, stored in context Δ'_d. Next, with this updated values and the transitioning port p as context, the destination state or barrier gets activated, followed by deactivation of the port p.

$$\text{PTrans1:}\ \frac{\begin{array}{l}\Delta \vdash s.active \rightsquigarrow \texttt{true} \\ \forall a_i \in s.actions:\ i = 1...n,\ \Delta \rightharpoonup \Delta^0 \\ \quad \Sigma, \Delta^{i-1} \vdash a_i \rightsquigarrow a_i, \Delta^i \ \rightharpoonup \Delta' \\ \left.\begin{array}{l}\forall (p,d) \in active\_ports(\Delta', s) \neq \emptyset:\ \Delta' \rightharpoonup (\Delta_p, \Delta_d) \\ \quad \forall pa \in d.parameters: \\ \quad\quad \Delta \vdash pa.expr \rightsquigarrow c \in \mathcal{C}, \Delta'_d[\Pi'] \\ \quad \Sigma, \Delta'_d, p \vdash d \longrightarrow_{\uparrow} d', \Delta''_d \\ \quad \Delta_p \vdash p \rightsquigarrow p'[active = \texttt{false}], \Delta'_p\end{array}\right\} \rightharpoonup \Delta''\end{array}}{\Sigma, \Delta \vdash s \longrightarrow s, \Delta''} \tag{3.28}$$

Again, *each* transition from ports p to destinations d gets evaluated independently, while the final context Δ'' is the cumulative result of *all* changed contexts Δ'_p and Δ''_d. Intuitively, this results in an observed (by upper layers) simultaneous transition of all active ports to their corresponding destinations. Furthermore, the rule is only applicable if there are any pending port-issued transitions, as indicated by $active\_ports(\Delta', s) \neq \emptyset$.

At first glance, there exists an issue with the independent transitioning semantics of PTrans1. When two ports with the same destination are active simultaneously, the destination contexts Δ_d are obviously not independent. In this special case, it is of importance, which of the 'conflicting' transitions is evaluated first, in order to determine the derivation sequence when applying rule PTrans1. Although this precedence selection is prone to the same issue pointed out for conflicting port conditions, the derivation trees are equivalent for this case, as the state does not get entered anymore when already active (see Section 5.2).

Barrier transition. Similarly to the transition semantics of activated ports, a deactivating barrier results also in transitions to all of its syntactically connected states. Let the function $active\_barriers : (\Delta \times \mathcal{S}) \rightarrow \mathcal{B}$ list all currently active barriers $b \in \mathcal{B}$ of a state $s \in \mathcal{S}$. Here, every active barrier results in the synchronous and thus independent activation of all its destinations. This is indicated by the specific destination contexts Δ_d, in which the destination states get activated after resolving eventual parameters (as in Rule PTrans1). Once all outbound states are activated the deactivation of the barrier element results in contexts Δ'_b. Finally, the semantic transition of *all* active barriers is captured in the final context Δ'' of state s, which also contains contexts of all, now active, destination states Δ''_d. Informally, the behavior of rule PTrans2 is, that *all* active child barriers transition to *all* their respective destination states at the same time. The premises of this rule are only fulfilled when at least one child barrier is active, i.e. $active\_barriers(\Delta, s) \neq \emptyset$.

$$\text{PTrans2:}\ \frac{\begin{array}{l}\Delta \vdash s.active \rightsquigarrow \texttt{true} \\ \forall a_i \in s.actions:\ i = 1...n,\ \Delta \rightharpoonup \Delta^0 \\ \quad \Sigma, \Delta^{i-1} \vdash a_i \rightsquigarrow a_i, \Delta^i\ \rightharpoonup \Delta' \\ \left.\begin{array}{l}\forall b \in active\_barriers(\Delta', s) \neq \emptyset:\ \Delta' \rightharpoonup \Delta_b \\ \quad \forall d \in b.out:\ \Delta' \rightharpoonup \Delta_d \\ \quad\quad \forall pa \in d.parameters: \\ \quad\quad\quad \Delta' \vdash pa.expr \rightsquigarrow c \in \mathcal{C}, \Delta'_d[\Pi'] \\ \quad\quad \Sigma, \Delta'_d, \emptyset \vdash d \longrightarrow_{\uparrow} d', \Delta''_d \\ \quad \Delta_b \vdash b \rightsquigarrow b'[active = \texttt{false}], \Delta'_b\end{array}\right\} \rightharpoonup \Delta''\end{array}}{\Sigma, \Delta \vdash s \longrightarrow s, \Delta''} \tag{3.29}$$

Stuttering

When a state $s \in \mathcal{S}$ is not subject to any progress (e.g. getting deactivated by a port, or a transition of its children), it may perform a semantic transition called *stuttering*. In this case, the state only evaluates its actions and propagates evaluation to active child states. Actions are evaluated sequentially, where propagation to child states is done synchronously (and independently) as described by the two different notations in the following rule.

$$\text{SProp:}\ \frac{\begin{array}{l}\Delta \vdash s.active \rightsquigarrow \texttt{true} \\ \forall a_i \in s.actions:\ i = 1...n,\ \Delta \rightharpoonup \Delta^0 \\ \quad \Sigma, \Delta^{i-1} \vdash a_i \rightsquigarrow a_i, \Delta^i\ \rightharpoonup \Delta' \\ \forall c \in active\_states(\Delta', s):\ \Delta' \rightharpoonup \Delta_c \\ \quad \Sigma, \Delta_c \vdash c \longrightarrow c', \Delta'_c\ \rightharpoonup \Delta''\end{array}}{\Sigma, \Delta \vdash s \longrightarrow s, \Delta''} \tag{3.30}$$

The propagation premise relation $c \longrightarrow c'$ is satisfied by all micro layer rules, as the transition includes also rules with conclusion $c \longrightarrow c$. Obviously in case of $s.actions = \emptyset$ and $s.children = \emptyset$ the resulting context Δ'' equals Δ. Summarizing, the application of rule SProp, recursively enables progress (preemption, transitions) or stuttering (including action evaluation) on all active children in the state-hierarchy, and eventually breaks down to a sequence of atomic layer rules.

If the *root state* $s_{root} \in \mathcal{S}$ at the top-level of this state-hierarchy progresses (i.e. transitions to s'_{root}), it will deactivate, preempt all active states and represent the end of the model execution. However, all micro steps get initiated by propagation (via rule SProp) of the root state. The rule capturing both these cases concludes in a semantic transition of the whole internal model state:

$$\text{Micro:}\ \frac{\Sigma, \Delta \vdash s_{root} \longrightarrow s'_{root}, \Delta'}{\Sigma \vdash \Delta \Longrightarrow \Delta'} \tag{3.31}$$

3.4.5 Quiescence

The repetetive application of micro steps, until the model state does not change anymore, reaching a *complete* state, is called *quiescence*. It is described by the relation $\Longrightarrow_{\downarrow}$ and defined using the transitive reflexive closure of the micro relation $\Longrightarrow^*$:

$$\text{Quiesc: } \frac{\Sigma \vdash \Delta \Longrightarrow^* \Delta' \quad \Delta' \; is \; \Longrightarrow\text{-normal-form}}{\Sigma \vdash \Delta \Longrightarrow_{\downarrow} \Delta'} \tag{3.32}$$

The semantics provided by this relation are also called *run-to-completion*, which hints on the complete state of the model reached after transitioning. A model state Δ' is called complete or in $\Longrightarrow$-normal-form [6], if it cannot be reduced any further by $\Longrightarrow$-transitions. This means, that there is no Δ'' such that $\Delta' \Longrightarrow \Delta''$.

3.4.6 Macro layer

In a given context Σ, the model Δ advances in a finite number of steps to a non-reducable state in micro steps, which is described by the quiescence relation. But in order to react on external stimuli from services, the context has to get updated with updated values from environment Ω. This incorporation of external information accompanied with the model reacting on this updated input is called a *macro step* and defined by the macro relation:

$$\Sigma' = \begin{cases} \Omega & \text{if } \mathit{updated}(\Omega, \Sigma) \\ \Sigma & \text{else} \end{cases}$$

$$\text{Macro: } \frac{\Sigma' \vdash \Delta \Longrightarrow_{\downarrow} \Delta'}{\Omega \vdash (\Sigma, \Delta) \ast\!\longrightarrow (\Sigma', \Delta')} \tag{3.33}$$

So far, the effects of interactions with services in the environment Ω while evaluating functions are not modeled, and thus, the *updated* function in the if-condition defining Σ', is modeled as the difference of Ω and Σ between two consecutive macro steps: $\mathit{updated}(\Omega, \Sigma) := \Omega \neq \Sigma$.

3.4.7 Execution

Using the macro relation, the entire execution can then be modeled as the application of macro steps. While neglected in lower layers, the changed environment Ω' is denoted here again to hint on possible interaction with the environment when evaluating functions on the atomic layer:

$$\text{Exec: } \frac{\Omega \vdash (\Sigma, \Delta) \ast\longrightarrow (\Sigma', \Delta'), \Omega'}{(\Omega, \Sigma, \Delta) \mapsto (\Omega', \Sigma', \Delta')} \tag{3.34}$$

In summary, the Exec rule describes an execution step of the RCM, assuming a changed environment Ω' without detailing these changes and how they would in turn affect the next macro step. This assumption can be overcome by modeling the interactions between evaluating model elements and the environment, as it is done in Chapter 4. Moreover, formalization of service interaction not only refines the formal semantics of RCM, but also enables formal analysis of tasks involving service operations.

Chapter 4

RACE-Service Model

In EMB.RACE, computational capabilities are modularized into RACE-Services, where the RACE-Service Model (RSM) serves two purposes: First, it describes an abstract syntactic interface for these services to uniformly interact with the various system capabilities from the RACE-Core. Secondly, it contributes to formalization of interaction between programs running in the RACE-Core (modeled by the RCM in Chapter 3) and respective service processes in the EMB.RACE ecosystem.

When researching component-based or service-oriented robotics, two different views can be identified. A *taxonomic* view is taken by analyzing typical application scenarios or similarly solved (robotic) problems, leading to a classification of services into a taxonomy. This advocates exchangability of components by solving the same problem, e.g. motion planning or object recognition. However, research is still in early stages on robotic service taxonomies [47], and only a small number of architectures approach modularization in this way (e.g. CLARAty [63]). The second and more popular approach, is to determine a component's compatibility by its *generic* interface, which often already describes component functionality. All stated component frameworks in Section 2.4 describe some sort of interface, which determines how modules can be interconnected.

The formalism presented in this chapter pursues a generic approach when modeling the syntactic interface of services after identifying primitive service modalities by analyzing typical robotic problems and their algorithmic solutions (taxonomic). On top of typical service-oriented features like re-usability, modularization and distributed operation of programs, RSM additionally enables *checked service coordination*, *assistance in orchestration* or *monitoring of execution* by formalizing service interaction.

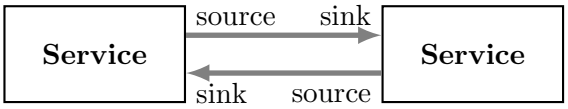
First, different kinds of service modalities are identified (Section 4.1), before introducing RSM syntax (Section 4.2), formal semantics (Section 4.3) and ultimately revisiting RCM syntax and semantics (Section 4.4) to additionally model service interaction.

4.1 Service modalities

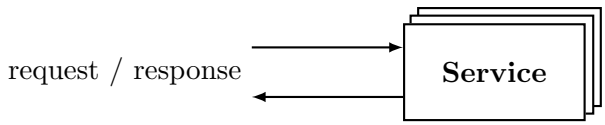
In the EMB.RACE system, RACE-Services represent the various system capabilities and provide a defined syntactic interface to coordinating entities like the RACE-Core, but also to other services. In order to specify requirements of this service interface, the employed communication modalities of typical robotic services have to be identified.

For this purpose, a variety of well-known robotic problems were analyzed in their required and provided data, parameterization and how they are utilized, i.e. sequential interaction with a particular service implementing a certain algorithm. Some of the considered scenarios are world modelling, motion planning, object recognition and visual servoing, graphical and haptic user-interfaces, grasping and robot control. After investigating these cases, two service communication primitives in three different modalities could be identified:

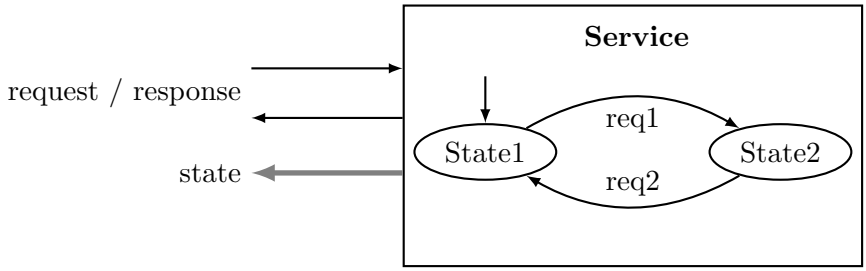
Data-flow. A service *publishes* or *subscribes* to data provided by another component. The transmitted data conforms to a specified data type and gets advertised with a certain frequency or even irregularly. As robotics is a very data-intensive domain, this type of communication is very common, e.g. processing pipelines of vision components. This modality is particularly interesting when feedback loops are present in the robotic system.



Request-response. This communication method corresponds to the classical *client-server* paradigm, where a service provides some functionality which can be invoked by other entities. The trivial case of such a service is the *stateless* variant, providing indempotent functionality which is always available and thus requests are handled immediately or at least independently.



In contrast to this, a *stateful* service provides state dependant functionality, possibly affecting service state on invocation. Most services in the robotic domain are long-running or resource-bound in one way or the other (computational time, physical movement or interaction, awaiting input, etc.) and are represeneted by this service modality.



In summary, the used communication primitives are well-known [41] and contained in most robotic software architectures and middlewares encountered so far. However, stateful components are seldomly distinguished and usually get reduced to their specified interface of data ports and procedures without any knowledge of the internal state [55]. The closest equivalent to modeling of stateful services are component life-cycles as they are defined in some frameworks (e.g. Orocos or SmartSOFT). These life-cycles typically specify whether a component is idle, active or in an error state.

4.2 RSM Syntax

This section introduces syntactic classes of RSM with the same notation as for RCM in Section 3.1. The primitive communication methods identified in Section 4.1 of *publish-subscribe* and *client-server* are used to specify the syntactic structure of a RACE-Service in RSM. The two different kinds of interaction paradigms are named *events* and *operations* respectively, to differentiate from middleware-specific nomenclature.

In general, RSM describes service states as finite state machines (FSM), using operation calls as transition events and extends these with notations to model deferred and pending responses (as explained later). A *service* $svc \in \mathcal{SVC}$ models stateful interaction by defining a generic service interface of events and operations, along with several internal states as follows.

Definition 4.1 (Service)**.**

$$\mathcal{SVC} = (\mathcal{ID}_{svc} \times \mathcal{OP}^* \times \mathcal{EV}^* \times \mathcal{T}^* \times \mathcal{SS} \times \mathcal{SS}) \tag{4.1}$$

The unique identifier of the service is given by $name(svc) \in \mathcal{ID}_{svc}$, where $operations(svc) \in \mathcal{OP}^*$ and $events(svc) \in \mathcal{EV}^*$ define the provided service interface. Transitions $trans(svc) \in \mathcal{T}^*$ describe possible changes of service states, while $init(svc) \in \mathcal{SS}$ specifies the initial and $cur(svc) \in \mathcal{SS}$ the currently active service state. For convenience, $op(svc, v) \in \mathcal{OP}^0$ and $ev(svc, v) \in \mathcal{EV}^0$ are defined to query operations and events by name.

Definition 4.2 (ServiceState)**.**

$$\mathcal{SS} = (\mathcal{V} \times \mathcal{EV}^*) \tag{4.2}$$

A service state $ss \in \mathcal{SS}$ has a locally unique $name(ss) \in \mathcal{V}$ along with $events(ss) \in \mathcal{EV}^*$ getting emitted in this state. The most practical granularity of specifying emitted events is still to be found as a more detailed specification similar to functions and actions of RCM states (Definition 3.1) would be possible as well.

Definition 4.3 (Transition)**.**

$$\mathcal{T} = (\mathcal{SS}^0 \times \mathcal{SS} \times \mathcal{OP}^0 \times \mathcal{EV}^* \times \mathcal{T}^*) \tag{4.3}$$

Transitions $t \in \mathcal{T}$ define an optional source state $src(t) \in \mathcal{SS}^0$, a required destination $dst(t) \in \mathcal{SS}$, the triggering $operation(t) \in \mathcal{OP}^0$ as well as emitted $events(t) \in \mathcal{EV}^*$ when transitioning. At last, syntactically defined relations $defer(t) \in \mathcal{T}^*$ are used to describe deferral of the transition's operation response to a set of other transitions.

Deferring an operation response basically means that the incoming operation call, which issues a state transition, is not answered directly, but will be resolved when some other transition is taken. This is consequently modeled by the auxiliary function *resolve*, which selects all deferred transitions getting resolved on transitioning of t:

$$resolve(t) = \begin{cases} x \mid \forall x \in \mathcal{T} & if \quad t.src = \emptyset, \\ x \mid \forall x \in \mathcal{T} : \ t \in x.defer & if \quad t.defer \neq \emptyset, \\ t \cup x \mid \forall x \in \mathcal{T} : \ t \in x.defer & else \end{cases}$$

Informally, the relation is defined by the following three cases: First, transitions without a source consider all transitions for resolving. In the second and third case, all transitions x deferring on t are included, where transition t is only contained when it does not defer resolution itself (third case).

Furthermore, the semantic value $pending(t) \in \{\texttt{true}, \texttt{false}\}$ describes whether the operation response of some transition t is still pending, and the function *respond* determines pending operations of corresponding transitions (to resolve):

$$respond(t) = \{x.operation \mid \forall x \in t.resolve : \ x.pending = \texttt{true}\}$$

In conclusion, transitions with an empty *src* state model an always possible transition to the given destination state, while a transition with empty *operation* do not require interaction to be taken. The intention of RSM is best described by an example, as shown in Figure 4.1. The simplified *Robot Service* provides three operations, namely

`stop`, `move` and `change_behavior`. The initial service state is *Idle*, where a call of operation `move` would result in a transition to *Moving*. The dashed connection of this transition to the one leading back into *Idle* is the graphical notation for a deferral of the `move` operation. Transitions for operation `change_behavior` do not issue a state change, while operation `stop` will always issue a transition to state *Idle* (and would respond any deferred transitions as stated earlier).

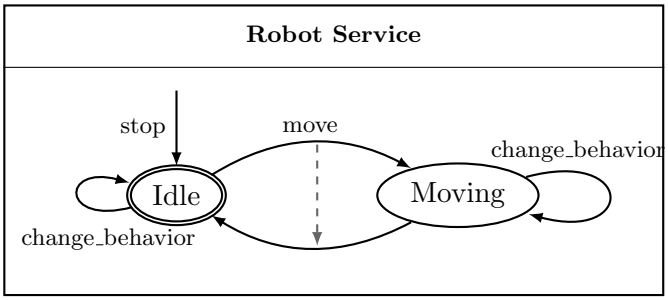


Figure 4.1: Simplified robot service providing operations `stop`, `move` and `change_behavior`, where *Idle* is the start state and the transition from *Idle* to *Moving* is deferred.

4.3 Formal semantics

As the syntactic structure of RSM is basically a FSM, the semantics correspond to finite state transducers. The intended behavior is that an RSM instance reacts on operation requests, possibly issuing state changes, and describes provided operation responses as well as published events of a service. By explicitly modeling available operations in each service state, the RSM semantics allow to decide if an issued sequence of operations can be executed (accepted) given a specific service model. On the other hand, the semantics also specify whether operation responses or event data is provided by a service in its current state, and permit statements about possible deadlocks or reachability of states.

4.3.1 Semantic domain and communication

The formal semantics of services are, similar as RCM in Section 4.3, defined by structured operational semantics (SOS) with labeled transitions systems (LTS) as semantic domain. The latter is defined as the four-tuple $(\mathcal{SVC}, L, \hookrightarrow, svc)$, where

- $\mathcal{SVC}$ are states of the transition system,

- $L = Com(\mathcal{OP})^0$ the labels,
- $\hookrightarrow \subseteq \mathcal{SVC} \times L \times \mathcal{SVC}$ is the semantic relation, and
- svc the initial semantic state.

As services are modeled to react on incoming operation calls, the formalism has to provide appropriate means of communication. Therefore, the labels of the service LTS use *channel systems* to synchronously receive operation calls from RCM [7, p. 53]. Channels provide both, synchronous and asynchronous communication via first-in/first-out buffers and are denoted by two actions defined in the set $Com = \{c!x, c?x\}$. Sending is realized with $c!x$, which transmits a value x via the channel $c \in Chan$, while $c?x$ consequently receives a value from the channel. A channel with capacity $size(c) = 0$ represents synchronous communication, which is also called *handshaking* or *rendezvous*.

Labels of the service LTS are then communicated operations of the form $\omega?\alpha$, where $\omega \in Chan$ and $\alpha \in \mathcal{OP}$ denote channel and operation respectively. In the RCM/RSM formalism, dedicated synchronous channels are obtained from the environment Ω via the function $channel(\Omega, svc) \in Chan$. Semantic transitions are written as $svc \overset{\omega?\alpha}{\hookrightarrow} svc'$ to represent an incoming operation call, on which the service should react, or $svc \hookrightarrow svc$ when no interaction occurs.

Once an operation request gets resolved by the service semantics, the corresponding response (either a result or error) or eventually published events will be available in the environment Ω. Mechanisms for providing and retrieving this information are realized by the two functions $get(\Omega, \epsilon) = c \in \mathcal{C}^0$ and $put(\Omega, \epsilon) = \Omega'$, where $\epsilon \in (\mathcal{OP} \cup \mathcal{EV})^*$ are requested operations or events. While the *get* function either returns the requested value or the empty set $\emptyset$ if unavailable, the *put* function produces a changed environment Ω', which contains the given operation response or event data ϵ. If ϵ contains multiple requirements, only available values are returned when queried via *get*. For example, let

$$\epsilon = (\texttt{'data'}, \texttt{true}) \in \mathcal{EV}$$

then an environment Ω can be accessed and changed by *get* and *put* as follows:

$$\begin{aligned} get(\Omega, \epsilon) &= \emptyset \\ put(\Omega, \epsilon) &= \Omega' \\ get(\Omega', \epsilon) &= (\texttt{'data'}, \texttt{true}) \end{aligned}$$

4.3.2 Semantic rules

Transitioning. In order to describe the transitioning of services, two different situations are distinguished. First, the service may perform a semantic transition (to a possibly different service state) when an operation request is received from the environment:

$$\text{Trans1:}\ \frac{\begin{array}{l}\exists t \in svc.trans : \\ \quad t.src = svc.cur \ \cup \ \emptyset \\ \quad \alpha = t.operation \quad \omega = channel(\Omega, svc) \\ \quad \epsilon = t.respond \ \cup \ t.events \\ \quad put(\Omega, \epsilon) = \Omega'[\epsilon]\end{array}}{\Omega \vdash svc \overset{\omega?\alpha}{\hookrightarrow} svc'[cur = t.dst], \Omega'[\epsilon]} \tag{4.4}$$

This rule considers both, transitions with and without a source state. The premises require a transition to originate from the current service state *svc.cur* (if applicable) and compare the transition trigger *t.operation* with the retrieved operation request α via channel ω. If the premises are fulfilled, the semantic transition concludes in a possibly different service state as stated by $svc'[cur = t.dst]$, and a changed environment Ω' holding the appropriate operation responses and emitted events as listed in ϵ.

$$\text{Trans2:}\ \frac{\begin{array}{l}\exists t \in svc.trans : \\ \quad t.src = svc.cur \ \cup \ \emptyset \\ \quad t.operation = \emptyset \\ \quad \epsilon = t.respond \ \cup \ t.events \\ \quad put(\Omega, \epsilon) = \Omega'[\epsilon]\end{array}}{\Omega \vdash svc \hookrightarrow svc'[cur = t.dst], \Omega'[\epsilon]} \tag{4.5}$$

Rule Trans2 models the second case where transitions, which do not specify an operation trigger *t.operation*, may be taken similarly even when no interaction occurs. These types of transitions typically describe a finished computation which was originally issued by an operation call and got deferred (e.g. the transition from *Moving* to *Idle* of the example shown in Figure 4.1).

Stuttering. When no operation request are received by Ω, the service may not change its current service state, where only specified events of the current state get emitted into the environment (can be $\emptyset$).

$$\text{Stutter:}\ \frac{\begin{array}{l}\epsilon = svc.cur.events \\ put(\Omega, \epsilon) = \Omega'[\epsilon]\end{array}}{\Omega \vdash svc \hookrightarrow svc, \Omega'[\epsilon]} \tag{4.6}$$

4.4 Service interaction in RCM

So far, the effects of interaction with services are modeled in the RSM formal semantics. In order to fully describe communication between coordinating (RCM) and computing (RSM) processes, the semantics of the former have to get extended with corresponding mechanisms as well. For this purpose, the syntactic structure of functions and conditions has to get refined, as the chosen abstraction (presented in Section 3.4) is too coarse to describe interaction semantics. Figure 4.2 gives an outline to the modeled interaction semantics, by highlighting three key aspects: First, *calling* an operation on function evaluation (atomic layer) directly influences services in the environment $\Omega_{svc} \subseteq \Omega$, possibly resulting in service state transitions. Conditions and functions *depend* on operation results or event data in the current external context Σ (invariant during a macro step). And as a third point, the external context gets *updated* with values from services in the environment Ω on every macro step.

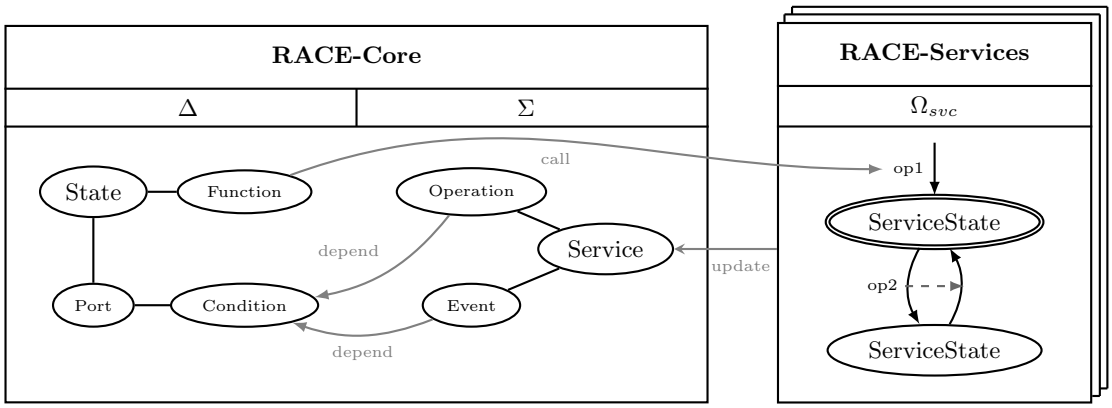


Figure 4.2: Schematic visualization of the service interaction formalization, where a function and condition in a RACE-Core instance interact with RACE-Services. The evaluation of a function with service interaction yields direct changes to Ω, while dependencies have to be met by external context Σ, which gets updated from Ω on every macro step.

4.4.1 Revisited RCM syntax

Boolean conditions are refined to distinguish between regular boolean expressions and service dependant portions of the condition. Parts of both kinds are later semantically combined by boolean operators, while the compound boolean condition is defined as:

Definition 4.4 (Boolean Condition)**.**

$$\mathcal{BC} = (\mathcal{BE} \cup \mathcal{BD})^* \tag{4.7}$$

$\mathcal{BE}$ are boolean expressions written in function language without effects on the model and may read available parameters, results or variable values of a given internal context.

$\mathcal{BD} = (\mathcal{ID}_{svc} \times \mathcal{V}^* \times \mathcal{V}^*)$ are boolean expressions depending on service event and operation result data. $svc(sd) \in \mathcal{ID}_{svc}$ specifies which service, while $ev(sd) \in \mathcal{V}^*$ and $op(sd) \in \mathcal{V}^*$ describe which events/operations are required to evaluate. Note that only event and operation names are defined by these realations, which will get resolved to respective instances. Also, expressions of this type may not have any effect on the model on evaluation.

Functions $f \in \mathcal{F}$ get redefined similarly as a compound of individual statements:

Definition 4.5 (Function)**.**

$$\mathcal{F} = (\mathcal{ST} \cup \mathcal{SD} \cup \mathcal{SC})^* \tag{4.8}$$

$\mathcal{ST}$ are statements written in a function language (like calculations, variable assignments, prints/logs etc.), with the constraint, that long-blocking statements are not allowed (e.g. sleep, I/O, synchronous communication etc.), but may use and *change* values of the current semantical context.

$\mathcal{SD} = (\mathcal{ID}_{svc} \times \mathcal{V}^* \times \mathcal{V}^*)$ are statements (e.g. assignment to a local variable) depending on service event or operation result data. Again, $svc(sd) \in \mathcal{ID}_{svc}$ holds the service identifier, while $ev(sd) \in \mathcal{V}^*$ and $op(sd) \in \mathcal{V}^*$ define the event and operation names of this dependency.

$\mathcal{SC} = (\mathcal{ID}_{svc} \times \mathcal{V})$ are statements interacting with a service by invoking an operation, where $svc(sc) = \mathcal{ID}_{svc}$ specifies the service, and $op(sc) = \mathcal{V}$ holds the name of the operation to call.

4.4.2 Revisited RCM semantics

Due to the syntactic refinement, the semantics of RCM consequently need to get adapted on the atomic, macro and execution layers. Besides of the redefined syntactic structure of functions and conditions requiring appropriate semantics, the macro and execution relations get detailed to additionally reflect service interaction.

Atomic layer

The semantics for the service dependant syntax elements $\mathcal{BD}$ and $\mathcal{SD}$ are described by LTS of the form $(\mathcal{BD}, L, \rightsquigarrow, bd)$, where the set of labels $L = (\mathcal{OP} \cup \mathcal{EV})^*$ is defined as sequences of operations or events and gets interpreted as service requirements. Intentionally, statements and expressions depending on certain events or operations, require corresponding values to be available on evaluation.

Service call statements $\mathcal{SC}$, on the other hand, are defined by an LTS with labels $L = Com(\mathcal{OP})$. Even though Com also includes the receiving communication action $c?x$, operation call semantics are only denoted with sending labels $c!v$, where the term $\omega!\alpha$ is used to describe an invocation of operation $\alpha \in \mathcal{OP}$ via channel $\omega \in Chan$.

Service lookup. All service relevant elements hold identifiers $id \in \mathcal{ID}_{svc}$ of services instances, which have to be resolved first in an external RCM context Σ. Note that an unvailable service will result in $svc = \emptyset$.

$$\text{Lookup: } \frac{}{\Sigma \vdash id \rightsquigarrow svc \in \mathcal{SVC}^0} \tag{4.9}$$

Boolean expressions. Service dependant boolean expressions $bd \in \mathcal{BD}$ evaluate into a boolean value, but require certain event or operation result/error data in order to do so. The premise $get(\Sigma, \epsilon) \neq \emptyset$ denotes this and holds when all requirements ϵ are present in the context Σ. This check is called *semantic availability* of these values, while the preceding premises test *syntactic availablity* of required events and operations. In case of $\mathcal{BD}$-elements, the requirements are stored in $bd.ev$ and $bd.op$.

$$\text{CondBD: } \frac{\begin{array}{l}\Sigma \vdash bd.svc \rightsquigarrow svc \neq \emptyset \\ \forall ev \in bd.ev : svc.ev(ev) \neq \emptyset \\ \forall op \in bd.op : svc.op(op) \neq \emptyset \\ \epsilon = (bd.ev \cup bd.op),\ get(\Sigma, \epsilon) \neq \emptyset\end{array}}{\Sigma \vdash bd \overset{\epsilon}{\rightsquigarrow} b \in \{\texttt{true}, \texttt{false}\}} \tag{4.10}$$

In contrast, boolean expressions which only use values of the internal context (e.g. parameters or results) evaluate without any side-effects or requirements to a boolean value:

$$\text{CondBE: } \frac{}{\Sigma \vdash be \rightsquigarrow b \in \{\texttt{true}, \texttt{false}\}} \tag{4.11}$$

Boolean conditions. The semantics for a boolean condition $bc \in \mathcal{BC}$ consisting of single boolean expressions $bc_i \in (\mathcal{BE} \cup \mathcal{BD})$ then accumulates requirements ϵ_i of its service dependant expressions as described by the following rule:

$$\text{Cond: } \frac{\begin{array}{c}\forall bc_i \in bc: \ i = 1...n, \\ \Sigma, \Delta \vdash bc_i \overset{\epsilon_i}{\rightsquigarrow} b_i \in \{\texttt{true}, \texttt{false}\}\end{array}}{\Sigma, \Delta \vdash bc \overset{\epsilon}{\rightsquigarrow} \texttt{bool}(bc, b_1, ..., b_n) \in \{\texttt{true}, \texttt{false}\}} \tag{4.12}$$

Here, $\epsilon = \bigcup_{i=1}^{n} \epsilon_i$ is the combination (union) of event/operation dependencies and $\texttt{bool} : \mathcal{BC} \times \mathcal{I}^* \rightarrow \mathcal{I}$ combines the results of bc_i using the boolean operators specified in the syntax of the boolean condition.

Service dependant statements. Similar to their condition counterpart, statements $sd \in \mathcal{SD}$ describe the dependency of a function statement on current semantic values of service events or operations, i.e. their semantic availablity. The only difference is, that these statements may additionally change the context Δ in its result Ξ or variable Λ values.

$$\text{FuncSD: } \frac{\begin{array}{c}\Sigma \vdash sd.svc \rightsquigarrow svc \neq \emptyset \\ \forall ev \in sd.ev : svc.ev(ev) \neq \emptyset \\ \forall op \in sd.op : svc.op(op) \neq \emptyset \\ \epsilon = (sd.ev \cup sd.op), \ get(\Sigma, \epsilon) \neq \emptyset\end{array}}{\Sigma, \Delta \vdash sd \overset{\epsilon}{\rightsquigarrow} sd, \Delta'[\Xi', \Lambda']} \tag{4.13}$$

Service operation call statements. These special statements $sc \in \mathcal{SC}$ express interaction with a syntacticly defined operation $sc.op$ of some service $sc.svc$. After resolving the service identifier and checking syntactic availability of the requested operation, the statement evaluates by synchronously invoking the operation α with the current environment Ω, via the appropriate channel ω of the requested service.

$$\text{FuncSC: } \frac{\begin{array}{c}\Sigma \vdash sc.svc \rightsquigarrow svc \neq \emptyset \\ \alpha = svc.op(sc.op) \neq \emptyset \\ \omega = channel(\Omega, svc)\end{array}}{\Omega, \Sigma \vdash sc \underset{\omega!\alpha}{\rightsquigarrow} sc} \tag{4.14}$$

The operation call itself does not change the environment directly, but will eventually result in a response from the service, which then changes Ω and results in available data in Σ of the following macro step.

The trivial case of function statements $st \in \mathcal{ST}$, which do not depend on service data or inflict operation calls, is described by the rule:

$$\text{FuncST: } \frac{}{\Sigma, \Delta \vdash st \rightsquigarrow st, \Delta'[\Xi', \Lambda']} \tag{4.15}$$

Functions. Since functions $f \in \mathcal{F}$ are composed of a sequence of individual statements, the semantics is defined similar to boolean conditions as the aggregated sequential evaluation of their statements.

$$\text{Func: } \frac{\begin{array}{c}\forall f_i \in f: \ i = 1...n, \ \Delta \rightharpoonup \Delta^0, \ \Omega \rightharpoonup \Omega^0 \\ \Sigma, \Delta^{i-1} \vdash f_i \underset{\omega!\alpha_i}{\overset{\epsilon_i}{\rightsquigarrow}} f_i, \ \Delta^i[\Xi', \Lambda'] \qquad \Omega^{i-1} \vdash svc \overset{\omega?\alpha_i}{\hookrightarrow} svc_i', \ \Omega^i[\epsilon_i']\end{array}}{\Omega, \Sigma, \Delta \vdash f \overset{\epsilon}{\rightsquigarrow} f, \ \Delta^n, \Omega^n[\epsilon']} \tag{4.16}$$

As the individual statements are $f_i \in (\mathcal{ST} \cup \mathcal{SD} \cup \mathcal{SC})$, three cases are distinguished:

- Service interacting statements $f_i \in \mathcal{SC}$ specify an operation to call $\alpha_i \neq \emptyset$ and the second premise on the right-hand side is applicable, which incorporates synchronous handshaking communication with the appropriate service process *svc*. These statments may not change the context, and thus $\Delta^i = \Delta^{i-1}$.
- Service dependant statements $f_i \in \mathcal{SD}$ expect a set of required event and operation data $\epsilon_i \neq \emptyset$ in order to get evaluated. On evaluation these statements may change results or variable values.
- Regular function statements $f_i \in \mathcal{ST}$ without requiring any data from services or issuing operation calls, but still change the context.

Obviously in the latter two cases, when $\alpha_i = \emptyset$, the environment is unchanged $\Omega^i = \Omega^{i-1}$. Again, $\epsilon = \bigcup_{i=1}^{n} \epsilon_i$ are requirements of individual function statements.

Informally, application of this rule evaluates each statement of a function in sequence, where service dependant functions require the appropriate event data or operation results/errors to be present in the context Σ. Conversely, service call statements communicate with services and ultimately result in certain event data getting published by corresponding services on or after interaction.

Actions. The semantics definition of action elements $a \in \mathcal{A}$ is adapted to account for propagated service requirements from its condition and function semantics:

$$\text{Action1: } \frac{\begin{array}{c}\Sigma \vdash executed(\Sigma, a) \rightsquigarrow \texttt{false} \\ \Sigma, \Delta \vdash a.cond \overset{\epsilon_c}{\rightsquigarrow} \texttt{true} \\ \Omega, \Sigma, \Delta \vdash a.func \overset{\epsilon_f}{\rightsquigarrow} a.func,\ \Delta', \Omega'\end{array}}{\Omega, \Sigma, \Delta \vdash a \overset{\epsilon_c \cup \epsilon_f}{\rightsquigarrow} a,\ \Delta', \Sigma'[executed \overset{\cup}{=} a], \Omega'} \tag{4.17}$$

$$\text{Action3: } \frac{\Sigma, \Delta \vdash a.cond \overset{\epsilon_c}{\rightsquigarrow} \texttt{false}}{\Sigma, \Delta \vdash a \overset{\epsilon_c}{\rightsquigarrow} a} \tag{4.18}$$

Activation and Micro layer

Relations of the activation and micro layer get also annotated by service requirements ϵ, each collecting these from their premises, similarly as boolean condition and function relations of the atomic layer. However, they have no effect on the semantics of the activation and micro layer, and thus, are not enumerated here for the sake of brevity.

Macro layer

On the macro layer, the modeling of service interaction allows to redefine the *updated* function to describe the changes necessary for an updated external context Σ more faithfully: $updated(\Omega, \Sigma) := get(\Omega, \epsilon) \neq get(\Sigma, \epsilon)$. Here, ϵ are the cumulative service dependencies (event or operation response data) of a macro step, which break down to all dependencies of functions and boolean conditions (atomic layer) getting evaluated in the quiescence premise of the macro relation. Hence, the model can only transition via the macro relation if all requirements ϵ are met in the context Σ'.

$$\Sigma' = \begin{cases} \Omega & \text{if } updated(\Omega, \Sigma) \\ \Sigma & \text{else} \end{cases}$$

$$\text{Macro: } \frac{\Sigma' \vdash \Delta \overset{\epsilon}{\Longrightarrow}_{\downarrow} \Delta', \Omega'}{\Omega \vdash (\Sigma, \Delta) \ast\!\longrightarrow (\Sigma', \Delta'), \Omega'} \tag{4.19}$$

Execution layer

Besides modeling execution of the core model and its effects on corresponding services, the Exec relation also captures progress of service processes on its own. As already outlined in the formal semantics of RSM in Section 4.3, services may take transitions not issued via communication or also emit events when stuttering (no change of service state). Formally, the synchronous (and independent) transition of services is stated in the following rule:

$$\text{Exec:}\ \frac{\begin{array}{l}\Omega \vdash (\Sigma, \Delta) \ast\!\!\longrightarrow (\Sigma', \Delta'), \Omega' \\ \forall svc_i \in \Omega' : \Omega' \rightharpoonup \Omega_{svc} \\ \quad \Omega_{svc} \vdash svc_i \hookrightarrow svc_i,\ \Omega'_{svc} \rightharpoonup \Omega''\end{array}}{(\Omega, \Sigma, \Delta) \mapsto (\Omega'', \Sigma', \Delta')} \tag{4.20}$$

Macro steps may issue service interaction by invoking operation calls and thus might inflict changes to the environment, producing Ω'. The independent semantic transitions of all services in changed environment Ω', result cumulatively in environment configuration Ω'', which also holds emitted events from stuttering or non-interaction transitions of relevant service processes. $\Omega' \rightharpoonup \Omega_{svc}$ emphasizes the fact that provided values of a service *svc* do not conflict with values of other services in environment Ω''.

In conclusion, service interaction can be statically checked for inconsistencies in the execution semantics, by enumerating service dependencies ϵ of a concrete execution model Δ, and checking their availability in environments Ω.

Chapter 5

Model Validation

This chapter elaborates on employed validation techniques of RCM/RSM syntax and semantics. In contrast to the approaches presented here, formal verification of the languages would require further translation of the formal semantics to input languages of model checking or automated theorem proving systems (e.g. Promela for SPIN [37] or Isar for Isabelle [50]) and thus is beyond the scope of this thesis. However, the following three validation approaches, already provide a fair estimate on the correctness and applicability of the employed formalisms.

- **Expressiveness of syntax.** Well known and often encountered coordination problems are realized using RCM syntax, where modeling techniques and expressiveness get analyzed and critically reflected. Considered concepts are concurrency, error handling and loops in different application scenarios, which were also used in a full demonstrative application (see Appendix A).
- **Intended and formal semantics.** Key points of coordination models are examined, where the intended behavior of an RCM instance is compared with derivation trees of applying formal operational semantics according to the modeled rules. The rigorous application of SOS rules allows for a very fine-grained analysis of individual evaluation steps of the execution semantics.
- **Service interaction.** Known problem scenarios, where faults can only be detected by considering service interaction of the coordination model, are taken into account. Scenarios of this kind are given by instances of RCM and RSM, where the interaction is examined by formally deriving the detailed sequence of operation calls and emitted events when executing the core model. One of the researched properties in these situations is reachability of states in core and service models.

5.1 Expressiveness of syntax

Composition

Besides examples given throughout this thesis, the first scenario in this section again reflects on the compositionality of RCM states. The composite state *ObjectRecognition* implements coordination of an object recognition and database storage service and is presented in Figure 5.1. The two services are referred to as `obj_rec`, which provides an event `recognized_objects`, and `db`, which provides operations to store and query data (e.g. `get` or `add` operations).

First, the database service is accessed to retrieve the extrinsic camera calibration stored as a transform from origin to the camera (`o_t_c`). Then, *DetectObject* waits for recognized objects and filters recognition results based on the parameterized `label` and `check_z`-flag. If no or the wrong objects are detected by the service, this state will not exit. Once a correct object is detected, the result `o_t_x` gets calculated and set. The next state *GetGraspPose* then uses this result (as parameterized) to calculate a graspable pose and saves it as own result `o_t_x`. Finally, *SaveObjectPose* uses the database service to store the pose at the configured `key`. *ObjectRecognition* exits via the *success* port if the pose was saved successfully (service `db` returned the `key` as result for operation `add`). On the other hand, the *error* port gets triggered when either loading or saving of the poses failed (port condition is omitted in figure).

Conclusion of *ObjectRecognition*:

- The example shows parameter passing from parent to children, but also parameterization by sibling results. The mechanism realizes a clear separation of scopes, where at first it might be a bit unintuitive as scopes of parameter expressions are different from the used scope in functions or actions.
- Parameterization also allows for setting constant values of parameters in states on all levels of the hierarchy. When re-using existing state hierarchies, these default values can be used as a starting point.
- On the other hand, results need to be set via actions. In spite of the laborious nature of this necessity, this also ensures that lower level functionality does not accidentally change values of upper layers, which was a mature issue in previous protoypes.

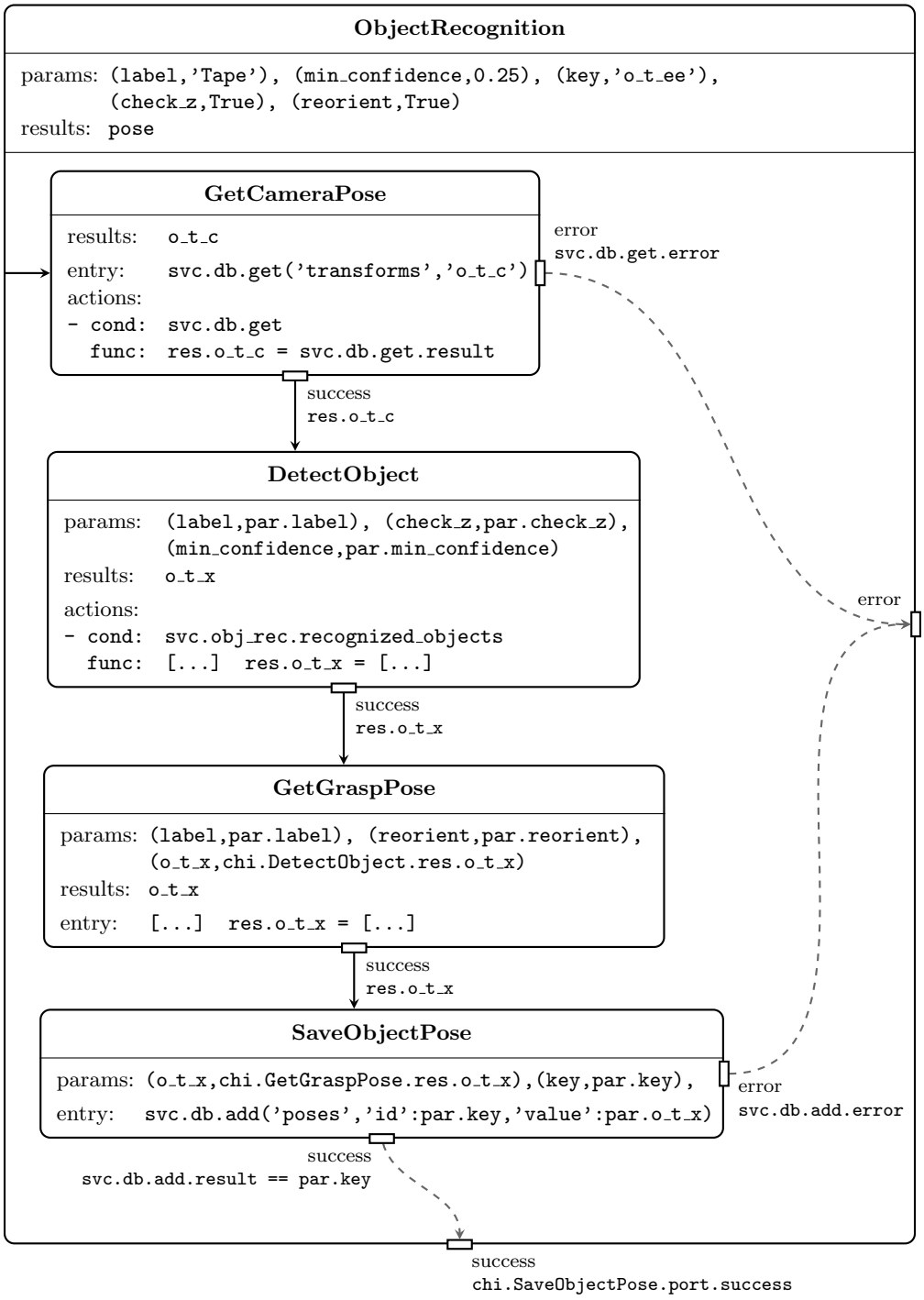


Figure 5.1: *ObjectRecognition* state comprised by a sequence of individual states. Actions of states *DetectObject* and *GetGraspPose* are not detailed any further and the notation expresses that the according result gets eventually set.

Decisions and fallback

A sophisticated (but still simplified) *Pick* state is outlined as an example for alternative logical flows in a state, as well as handling errors with contingency (self-correcting). Figure 5.2 shows the graphical representation of this coordination scenario. Most of the functionality, like grasping and approaching, is abstracted away in this example to keep focus on the alternative branching of coordination. Also note that the *Pick* error port is also triggerd on *TeachPose* error, but not shown connected to keep the visualization simple.

Pick is parameterized by a variety of approaching and grasping relevant values, where the most important parameter `pose` is used to query the object pose to pick from the database service (as in the last example). Consequently, the functionality of this example complements the *ObjectRecognition* state and can be used to pick a pose determined by object recognition.

After opening the gripper via the corresponding service in state *Release*, the *GetPose* state queries said object pose from the database service. At this point, a conservative implementation of *Pick* has to abort via the *error* port in case the specified pose should not be available in the database. However, the presented scenario tries to mitigate this fault by activating the *TeachPose* state in this case. It is assumed here that the *TeachPose* state is implemented to instruct the operator to move the robot in gravity compensation to the desired position and saves this pose to the database. Conversely, when the pose is available in the first place, the robot would approach it in the *Approach* state. Both alternative execution paths result in the state *Grasp*, which simply closes the gripper to the desired `width` and fails if no part is detected. In the latter case the *Pick* should not abort as well, but should retreat the robot first. This is achieved by transitioning also from the *error* port to *Retreat*. Finally, the state *CheckPart* decides whether the pick was successful or not.

Conclusion of *Pick*:

- The presented model allows for different paths depending whether the pose is already known (successfully retrieved from database) or not. In this case, this enables to take alternative action and correct foreseeable errors.
- Similarly, the transition from *Grasp*'s error port, allows to fall back on unexpected errors (e.g. due to the variable environment). However, this could be also implemented differently as an action with condition `chi.Grasp.port.error` and function `res.success = False` which could in turn activate the error port of *Pick*.
- All in all, this shows the provided flexibility in handling errors or unexpected situations with port and action mechanisms.

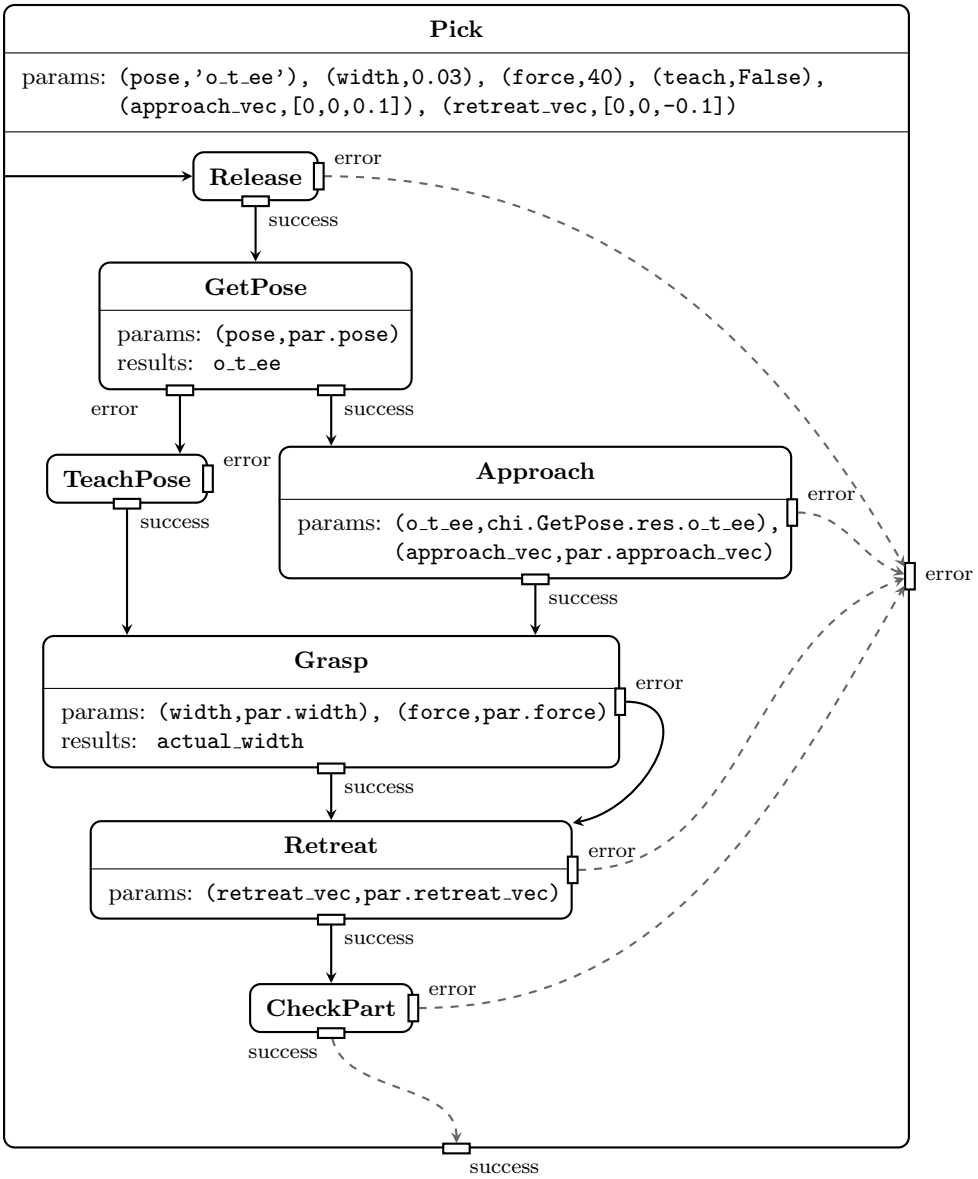


Figure 5.2: Example implementation of *Pick* which falls back on teaching if a pose should be unavailable

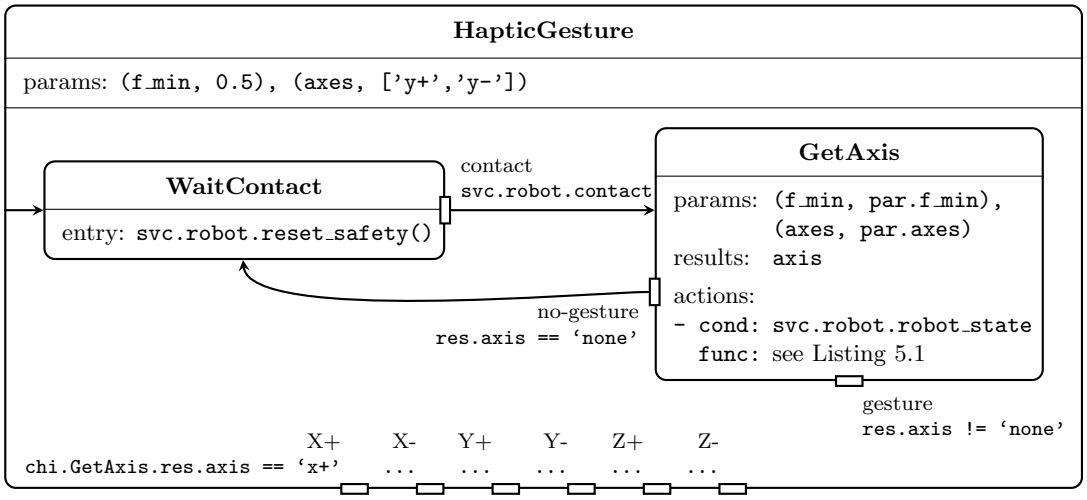


Figure 5.3: *HapticGesture* state, which loops until a configured minimal force `f_min` is detected on any of the parameterized `axes`

Iteration

The *HapticGesture* composite state is a typical example, where two states are looped until a certain input is detected by the robot, and is shown in Figure 5.3. In this case, the first state *WaitContact* listens on a contact event (`svc.robot.contact`), after which, external force measurement of the robot state gets extracted and interpreted in state *GetAxis*. If the detected force on the robot is too low or is not in the parameterized `axes`, the process restarts by transitioning into *WaitContact* again. Once the minimum force `f_min` in a configured axis is detected, the *GetAxis* state provides the result `axis`, which exits *HapticGesture* with the corresponding port. Additionally, this example showcases the function language by depicting the function content of *GetAxis*' action in Listing 5.1.

Conclusion of *HapticGesture*:

- This example shows a while-like loop, where *GetAxis* provides a sophisticated example of processing event data in a function in order to decide on iteration or termination of the loop.
- Also, the state *HapticGesture* is a good example for emphasizing the explicit interface of RCM states. It's two parameters `f_min` and `axes` fully specify its functionality and even restrict possible outcomes (port activations). The latter is convenient practice to dynamically specify state behavior without changing its syntactic structure, i.e. the number of ports is fixed for all instantiations of *HapticGesture* (should it get re-used).

Listing 5.1: Function code of *GetAxis* action

```
f_ext_hat = svc.robot.robot_state.f_ext_hat
axis = ['x', 'y', 'z']
f_max = 0
axis_max = 'none'
for i in range(len(f_ext_hat)):
    if axis[i] in par.axes:
        f = f_ext_hat[i]
        if abs(f) > par.f_min and abs(f) > f_max:
            f_max = abs(f)
            if f > 0:
                axis_max = axis[i] + '+'
            else:
                axis_max = axis[i] + '-'
res.axis = axis_max
```

Concurrency

The *SearchObject* state composes the previous *ObjectRecognition* and *HapticGestures* example states to additionally abort the object recognition by haptic interaction. Besides containing concurrency, this example also shows how results are set on a composite state before exiting and is depicted in Figure 5.4.

The barrier as first element enables both, *ObjectRecognition* and *HapticGestures* concurrently, where the functionality of these states is explained in the corresponding examples. Note that parameterization of *SearchObject* is not modeled to highlight, that constant (default) values of parameters can be specified on any level of the state hierarchy (but have to be resolvable on activation). In this case, this fixes *ObjectRecognition* to recognize objects with `label=='Tape'`, while *HapticGesture* only reacts on haptic interaction along the `z-` axis. Should *ObjectRecognition* find a `'Tape'` object, the composite state stores the pose result as its own result via an action reacting on `chi.ObjectRecognition.res.pose`, before triggering port *success*. This practice relies on the fact that results are written before relevant ports get triggered by their condition. Should an error occur while detecting the object the *SearchObject* state will exit on the *error* port, while alternatively, the human operator can abort the process by issuing a haptic gesture along the negative z axis. In both cases, all states in *SearchObject* are deactivated, no matter what caused deactivation of the composite state.

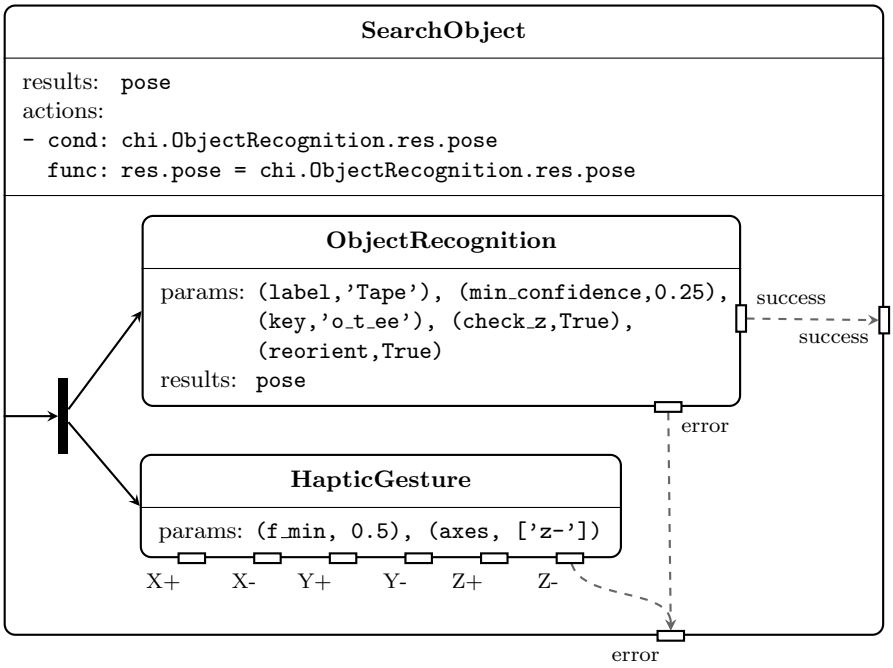


Figure 5.4: *SearchObject* state which is composed of other exemplary states *ObjectRecognition* and *HapticGesture* to model preemptive gestures while detecting objects

Conclusion of *SearchObject*:

- The *SearchObject* state models two simultaneously active states which alternatively result in fulfillment of port conditions, causing both states to deactivate. This resembles an OR-case of composing states, where an AND-composition would require a barrier which synchronizes both concurrent paths (constructed examples can be found in Section 5.2).
- Most common use of this syntax is to provide multiple interaction mechanisms, where each of them can be used alternatively to confirm or abort certain tasks.

5.2 Intended and formal semantics

This section elaborates on selected examples to highlight cases where execution semantics may not be straight-forward from intention alone. First, the scenario is described by the graphical representation of the state at hand with some additional information on assumed external communication or when certain ports activate in order to reach the discussed situation. Then, the intended behavior is stated and validated using formal execution semantics, by applying the respective SOS rules and building a derivation tree. For the sake of breviety, only the first couple of examples are fully derived here, where the remaining derivation trees can be found in Appendix B.

Action execution on state preemption

The first scenario investigates the state preemption mechanism by answering the question whether state functions get preempted when a port activates, where the examplary state *SumTestData* is shown in Figure 5.5.

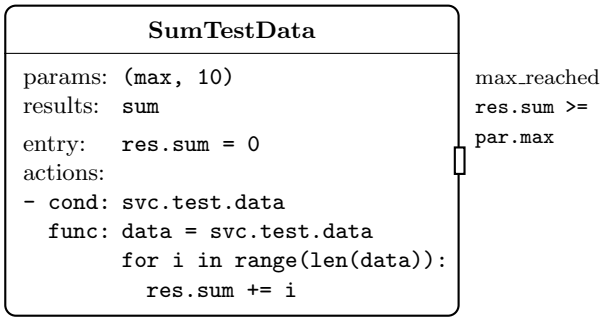


Figure 5.5: State with enabled port condition during action execution

Scenario:

- Two `data` events of service `test` are received in macro step 1 and 2
- Event data has length `8`, leading to the fulfilled port condition in macro step 2
- When is the state *SumTestData* preempted? Is the function executed completely?

Intention:

- When the first event is received, the result `sum` gets increased to a final result of `8`
- On the second event, the action gets executed again, increasing the value of `res.sum` to `16`
- The port condition is fulfilled, and state *SumTestData* gets deactivated

Formal derivation:
The environments Ω_0 and Ω_1 hold the first and second data event respectively, which leads to two macro steps in this execution. It is assumed that the state *SumTestData*, named s hereafter, is already entered and activated when the first macro step is processed. The single port *max_reached* is denoted as p.

Execution sequence of two Exec/Macro steps:

$$(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1', \Delta_1) \mapsto (\Omega_2, \Sigma_2', \Delta_2)$$

$$\frac{\frac{\dots}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1', \Delta_1), \Omega_1}}{(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1', \Delta_1)}$$

$$\frac{\frac{\dots}{\Omega_1 \vdash (\Sigma_1', \Delta_1) \ast\!\!\longrightarrow (\Sigma_2', \Delta_2), \Omega_2}}{(\Omega_1, \Sigma_1', \Delta_1) \mapsto (\Omega_2, \Sigma_2', \Delta_2)}$$

The first macro step consists of only one micro step, namely stuttering and action execution as defined in rule SProp.

Macro 1:

$$\frac{\Sigma_1 = \Omega_0 \qquad \frac{\frac{\dots}{\Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta_1, \Sigma_1'}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow_{\downarrow} \Delta_1, \Sigma_1'}}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1', \Delta_1)}$$

Micro 1-1:

$$\frac{\Delta_0 \vdash s.active \rightsquigarrow \texttt{true} \qquad \frac{\frac{\dots}{\Sigma_1, \Delta^0 \vdash a_1 \rightsquigarrow a_1, \Delta^1, \Sigma_1'}}{a_1 \in s.actions \ \Delta_0 \rightharpoonup \Delta^0,\ \Delta^1 \rightharpoonup \Delta_1} \qquad active\_states(\Delta_1, s) = \emptyset}{\frac{\Sigma_1, \Delta_0 \vdash s \longrightarrow s, \Delta_1, \Sigma_1'}{\Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta_1, \Sigma_1'}}$$

$$\frac{\begin{array}{l}\Sigma_1 \vdash executed(\Sigma_1, a_1) \rightsquigarrow \texttt{false} \\ \Sigma_1, \Delta^0 \vdash a_1.cond \rightsquigarrow \texttt{true} \\ \Sigma_1, \Delta^0 \vdash a_1.func \rightsquigarrow a_1.func, \Delta^1[\texttt{res.sum = 8}]\end{array}}{\Sigma_1, \Delta^0 \vdash a_1 \rightsquigarrow a_1, \Delta^1, \Sigma_1'[executed \overset{\cup}{=} a_1]}$$

The second macro step starts with a very similar micro step (left out as the same rules are applied), but the result `sum` holds value `16` after evaluation. As a consequence, the premises of micro rule PTrigger are fulfilled, and the quiescence relation expands into a second micro step, which activates the port and consequently exits state s.

Macro 2:

$$\frac{\Sigma_2 = \Omega_1 \qquad \dfrac{\dfrac{\ldots}{\Sigma_2 \vdash \Delta_1 \Longrightarrow \Delta', \Sigma_2'} \quad \dfrac{\ldots}{\Sigma_2' \vdash \Delta' \Longrightarrow \Delta_2}}{\Sigma_2 \vdash \Delta_1 \Longrightarrow_{\downarrow} \Delta_2, \Sigma_2'}}{\Omega_1 \vdash (\Sigma_1', \Delta_1) \ast\!\!\longrightarrow (\Sigma_2', \Delta_2)}$$

Micro 2-2: Note that the *s.active* check is omitted due to the limited space

$$\frac{\Sigma_2', \Delta \vdash p.cond \rightsquigarrow \texttt{true} \quad \dfrac{\ldots}{\Sigma_2', \Delta' \vdash s \longrightarrow_{\downarrow} s', \Delta'''} \quad \dfrac{\Delta''' \vdash p.active \rightsquigarrow \texttt{false}}{\Delta''' \vdash p \rightsquigarrow p'[active = \texttt{true}], \Delta_2} \quad p \in s.ports}{\Sigma_2' \vdash \Delta' \Longrightarrow \Delta_2}$$

Deactivation of s with *s.active* check omitted:

$$\frac{s.children = \emptyset \qquad \Sigma_2', \Delta' \vdash s.exit \rightsquigarrow s.exit, \Delta'' \qquad \dfrac{\Delta'' \vdash s.active \rightsquigarrow \texttt{true}}{\Sigma_2', \Delta'' \vdash s \rightsquigarrow s'[active = \texttt{false}], \Delta'''}}{\Sigma_2', \Delta' \vdash s \longrightarrow_{\downarrow} s', \Delta'''}$$

In the final model state Δ_2, state s got deactivated after executing its action a second time, thus with result `res.sum = 16`, and port p is active.

Transitions on state preemption

The second example considers the case when a parent port depends on activation of a connected child port (as seen in Figure 5.6). The execution semantics have to define whether the transition is processed or preemption by activation of the parental port precedes.

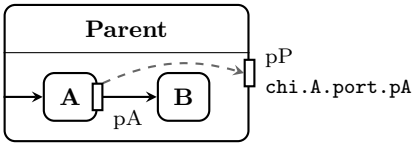


Figure 5.6: Scenario which investigates port activation of composite state *Parent* when a connected child port gets activated

Scenario:

- Port p_A gets activated in macro step 1 and is connected to state B
- Parental port p_P gets triggered on activation of p_A

Intention:

- Upon activation of p_A, state A gets deactivated first
- p_P gets enabled, which deactivates and preempts state *Parent*

Formal derivation:
It is assumed that states *Parent* and A, referred to as s_P and s_A, are already active when evaluationg macro step 1, which leads to a single execution step.

Execution:

$$\frac{\frac{\dots}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1)}}{(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1, \Delta_1)}$$

The macro step consists of two micro steps: First s_P propagates to its active child s_A, as no transitions are pending, and port p_A activates by application of rule PTrigger.

Macro 1:

$$\frac{\Sigma_1 = \Omega_0 \qquad \dfrac{\dfrac{\dots}{\Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta'} \quad \dfrac{\dots}{\Sigma_1 \vdash \Delta' \Longrightarrow \Delta_1}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow_{\downarrow} \Delta_1}}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1)}$$

Micro 1-1:

$$\frac{\dfrac{\Delta_0 \vdash s_P.active \rightsquigarrow \texttt{true} \quad s_P.actions = \emptyset \quad s_A \in active\_states(\Delta_0, s_P) : \ \dfrac{\dots}{\Sigma_1, \Delta_A \vdash s_A \longrightarrow s'_A, \Delta''_A \rightharpoonup \Delta'} \ \Delta_0 \rightharpoonup \Delta_A}{\Sigma_1, \Delta_0 \vdash s_P \longrightarrow s_P, \Delta'}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta'}$$

PTrigger of s_A, where the active check and exit function execution is omitted. For details on state deactivation $s_A \longrightarrow_{\downarrow} s'_A$ see the last example. Also, $[active]$ is used instead of $[active = \texttt{true}]$.

$$\frac{\Sigma_1, \Delta_A \vdash p_A.cond \rightsquigarrow \texttt{true} \quad \dfrac{\dots}{\Sigma_1, \Delta_A \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta'_A} \quad \dfrac{\Delta'_A \vdash p_A.active \rightsquigarrow \texttt{false}}{\Delta'_A \vdash p_A \rightsquigarrow p'_A[active], \Delta''_A} \quad p_A \in s_A.ports :}{\Sigma_1, \Delta_A \vdash s_A \longrightarrow s'_A, \Delta''_A}$$

In the second macro step, the parent state s_P does **not** transition candidate ports, because rule PTrigger has higher priority than PTrans1 (definition sequence).

Micro 1-2: Is then PTrigger of s_P, as the condition of port p_P evaluates to true ($s_P.active$ check is omitted).

$$\dfrac{\dfrac{\Sigma_1, \Delta' \vdash p_P.cond \rightsquigarrow \texttt{true} \quad \dfrac{\cdots}{\Sigma_1, \Delta' \vdash s_P \longrightarrow_{\downarrow} s'_P, \Delta''''} \quad \dfrac{\Delta'''' \vdash p_P.active \rightsquigarrow \texttt{false}}{\Delta'''' \vdash p_P \rightsquigarrow p'_P[active], \Delta_1}}{p_P \in s_P.ports :}}{\dfrac{\Sigma_1, \Delta' \vdash s_P \longrightarrow s'_P, \Delta_1}{\Sigma_1 \vdash \Delta' \Longrightarrow \Delta_1}}$$

Deactivation of s_P, with four premises getting evaluated as indicated by the contexts, propagating up with $(\Delta'_A, \Delta'_B) \rightharpoonup \Delta''$. The fourth premise, which sets $s.active$ to `false` and results in context Δ'''', is omitted in this derivation step to focus on the sequence of child deactivation and exit-function evaluation.

$$\dfrac{\dfrac{\cdots}{\Sigma, \Delta_A \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta'_A} \quad \dfrac{\cdots}{\Sigma, \Delta_B \vdash s_B \longrightarrow_{\downarrow} s'_B, \Delta'_B} \quad \Sigma_1, \Delta'' \vdash s.exit \rightsquigarrow s.exit, \Delta'''}{s_A, s_B \in s_P.children : \Delta' \rightharpoondown (\Delta_A, \Delta_B)}}{\Sigma_1, \Delta' \vdash s_P \longrightarrow_{\downarrow} s'_P, \Delta''''}$$

Deactivation of the children s_A and s_B is then done by applying rule SDeact1 to the already inactive states, but potentially active ports (p_A) get deactivated to preempt transitions. As the deactivation of s_B is trivial, only $s_A \longrightarrow_{\downarrow} s'_A$ is included here.

$$\dfrac{\Delta_A \vdash s_A.active \rightsquigarrow \texttt{false} \quad \dfrac{\dfrac{\Delta^0 \vdash p_1.active \rightsquigarrow \texttt{true}}{\Delta^0 \vdash p_1 \rightsquigarrow p'_1[active = \texttt{false}], \Delta^1 \rightharpoonup \Delta'_A}}{p_A = p_1 \in s.ports : \ \Delta_A \rightharpoondown \Delta^0}}{\Sigma, \Delta_A \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta'_A}$$

This derivation tree (especially the last expansion seen above) clearly shows, that all active ports and barriers are preempted when deactivating the parent state.

Action execution sequence

The scenario presented in this subsection elaborates on the sequence of action execution, where the constructed situation is depicted graphically in Figure 5.7. Considered issues are whether execution of an action does precede the transition from p_A to B or similarly if the second action is executed before or after B's entry function.

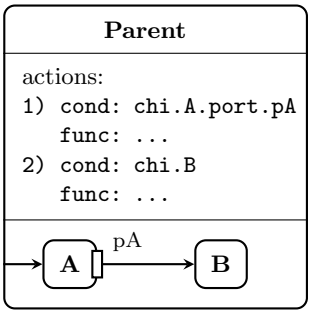


Figure 5.7: Exemplary state used for researching action execution sequence relative to transitioning semantics

Scenario:

- Port p_A gets activated in macro step 1 and is connected to state B
- Two actions a_1 and a_2 react on activation of p_A and B respectively
- In what sequence are these actions executed relative to the entry function of B?

Intention:

- After p_A got activated, the corresponding action has a fulfilled condition and executes, before B gets activated as destination of port p_A
- Upon activation of B, the entry function gets executed, while a_2 only executes after B was activated

Formal derivation:
Even though the formal derivation tree of executing this scenario is pretty straightforward, only the sequence of micro steps is included here for the sake of breviety. The full derivation tree of this and the following scenarios can be found in Appendix B. From the scenario it is assumed that the parent state s_P and state s_A are active when evaluating the first macro step.

Execution with one macro step:

- Micro 1-1: SProp of s_P, with no action conditions fulfilled; PTrigger of s_A, evaluating *exit* function of s_A and setting p_A active
- Micro 1-2: PTrans1 of s_P, where a_1 gets evaluated due to `chi.A.port.pA` evaluating to `true`, and a transition from p_A to B gets taken, which executes the *entry* function of B and ultimately activates B
- Micro 1-3: SProp of s_P, with a_2 being executed because `chi.B` is fulfilled; SProp of s_B with no further action evaluations or progress; quiescence reached

The formal derivation into three consecutive micro steps, clearly shows that the sequence of function evaluation conforms to the intention stated above.

Concurrent port activation (same destination)

The first example coping with concurrent port activation, describes the situation where ports with simultaneously fulfilled conditions have the same destination state. The configuration visualized in Figure 5.8 is also mentioned when defining micro layer rules in Section 3.4.

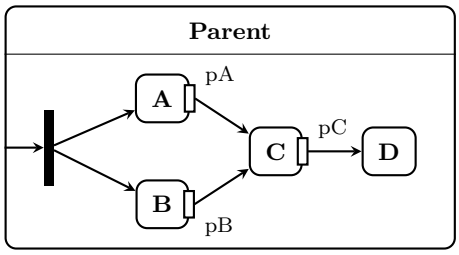


Figure 5.8: Composite state with concurrently activating child ports (same condition) and identical destination

Scenario:

- Ports p_A and p_B activate in macro step 2 and point to the same destination C
- The condition of p_C is fulfilled in macro step 3

Intention:

- At first, the barrier activates A and B
- As both ports p_A and p_B are active simultaneously, B gets activated by transitioning any of these two active ports
- Finally, when p_C activates, D gets activated and is the only active state

Formal derivation:
With s_P as the composite state, barrier b and states s_A, s_B, s_C and s_D, the execution of this scenario is described by three macro steps, where the first macro step starts with the activation of s_P.

Execution with three macro steps:

- Micro 1-1: SAct1 of s_P, which also activates first child b
- Micro 1-2: PTrans2 of s_P transitioning active barrier b to s_A and s_B, before quiescence is reached.

- Micro 2-1: SProp of s_P, where PTrigger is applicable to s_A and s_B as the respective conditions are fulfilled
- Micro 2-2: PTrans1 of s_P with two pending transitions from p_A to s_C and p_B to s_C. Evaluation order is not constrained and typically determined by order of states s_A and s_B in the *children* set of s_P. Assumining the transition originating from p_A is evaluated first, the destination state s_C gets activated via rule SAct1. The transition from p_B evaluated afterwards, results in application of rule SAct2, as state s_C is already active, and quiescence is reached.
- Micro 3-1: SProp of s_P and PTrigger of s_C as port condition of p_C is fulfilled
- Micro 3-2: PTrans1 of s_P as port p_C is active. After transitioning to state s_D quiescence is reached and the exemplary execution is finished

In case of concurrently active ports, the evaluation order of the pending transitions does not matter as the activation of states is not affected by the cause of activation. Even though barriers do keep track of the originating port when transitioning, this also is not affected by the sequence in which inbound ports are transitioned.

Concurrent port activation (different destinations)

The second example researches concurrent port activation with different destination states, but with consideration of state preemption. The scenario is depicted in Figure 5.9 and is comprised by a similar setup as the previous example. This time, however, the ports p_A and p_B have different destination states, while a parental port condition depends on activation of one of these destination states.

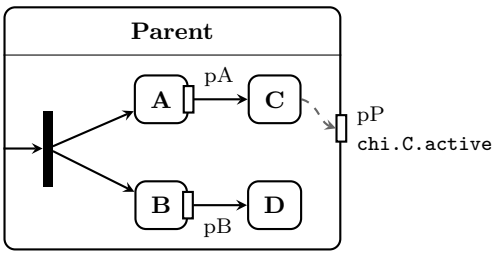


Figure 5.9: Composite state with simultaneously active child ports, transitioning to C and D, but preempting upon activation of C

Scenario:

- Ports p_A and p_B activate both in macro step 2
- Parental port p_P triggers on activation of state C
- Will D get entered before p_P preempts all active states in Parent? Does it depend on evaluation order of transitions from p_A and p_B?

Intention:

- At first, the barrier activates A and B
- Both, p_A and p_B activate states C and D respectively
- Only after both transitions were taken, the port p_P will get evaluated and preempts

Formal derivation:
The formal execution tree of state s_P in this scenario starts off very similar to the previous example and is described by two macro steps in total.

Execution with two macro steps:

- Micro 1-1: SAct1 of s_P, which also activates first child b
- Micro 1-2: PTrans1 of s_P transitioning from b to s_A and s_B, before quiescence is reached.
- Micro 2-1: SProp of s_P, where PTrigger is applicable to s_A and s_B as the respective conditions are fulfilled
- Micro 2-2: PTrans1 of s_P with two pending transitions from p_A to s_C and p_B to s_C. In this scenario, the unconstrained evaluation order results in two possible sequences in which states s_A and s_B get activated. However, at the end of this micro step the states are active in both alternatives.
- Micro 2-3: PTrigger of s_P, as the condition of port p_P evaluates to `true`. Quiescence is reached after preemption of both active child states s_A and s_B.

The formal sequence of taken steps clearly shows that also in this case evaluation order does not influence execution semantics and state s_D gets activated before preemption by port p_P takes place.

5.3 Service interaction

This section examines configurations of RCM and RSM intances, where faults like unreachable states or unavailable operations are detectable by formally deriving the individual steps of executing an RCM instance in a given scenario of services described by RSM instances.

Deadlock and Reachability

This example scenario elaborates on the detection of deadlocks in service interaction. Here, the constructed RCM and RSM instances eventually result in state where coordination cannot progress any further because it waits for a service response which will never get issued. Conversely, certain RCM states are not reachable in this situation, which hints on the violated *reachability* property of the RCM instance. Figure 5.10 shows the exemplary RCM and RSM instances of this scenario.

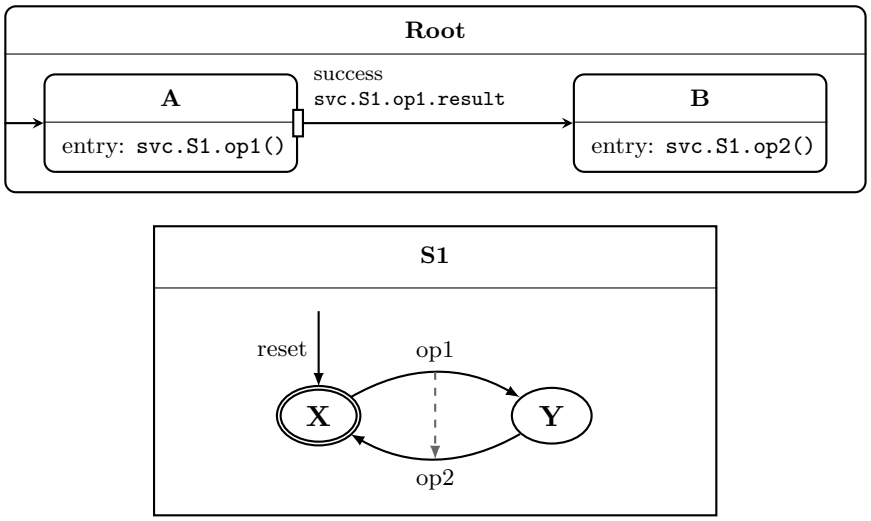


Figure 5.10: Deadlock example, where the RCM (top) and RSM (bottom) instances get stuck and cannot progress any further

Scenario:

- *S1* is in state *X* and *Root* activates in the first macro step
- *Root* is the only entity interacting with service *S1* (closed world assumption).
- The *first* state *A* invokes operation *op1* of service *S1*, where *A* is left when condition `svc.S1.op1.result` is fulfilled, which requires response to the invoked operation

- Service *S1* only responds to the deferred transition of operation *op1* when operations *op2* or *reset* are invoked
- Is state B reachable?

Intention:

- State A activates and issues call of operation $op1$, which changes service state to Y and defers response
- The *success* port of A only triggers with $op1$ getting returned, but as the transition is deferred, a response is only generated when $op2$ or $reset$ get invoked
- The configuration cannot progress any further and state B is not reachable

Formal derivation:
The states *Root*, A and B are referred to as s_R, s_A, s_B, where *S1* and its service states are denoted as svc_1, ss_X and ss_Y. The transition from ss_X to ss_Y triggering on $op1$ is referred to as t_1. It is assumed that the first micro step in the first macro step is the initial activation of s_R.

Execution 1: The first execution step consists of a macro step and a service transitition.

$$\frac{\dfrac{\dots}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1), \Omega'} \qquad \dfrac{\dfrac{\dots}{\Omega_{svc} \vdash svc_1 \hookrightarrow svc_1,\ \Omega'_{svc}}}{svc_1 \in \Omega' : \Omega' \rightharpoonup \Omega_{svc} \quad \Omega'_{svc} \rightharpoonup \Omega_1}}{(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1, \Delta_1)}$$

Macro 1: As already stated, the first macro step is assumed to just consist of a micro step which activates s_R.

$$\frac{\Sigma_1 = \Omega_0 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Omega_0, \Sigma_1, \Delta_0, \emptyset \vdash s_R \longrightarrow_{\uparrow} s'_R, \Delta_1, \Omega'}}{\Omega_0, \Sigma_1, \Delta_0 \vdash s_R \longrightarrow s'_R, \Delta_1, \Omega'}}{\Omega_0, \Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta_1, \Omega'}}{\Omega_0, \Sigma_1 \vdash \Delta_0 \Longrightarrow_{\downarrow} \Delta_1, \Omega'}}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1), \Omega'}$$

Activation of s_R, with omitted $s_R.active$ check

$$\frac{\dfrac{\Delta_0 \vdash s_R.active \rightsquigarrow \texttt{false}}{\Delta_0 \vdash s_R \rightsquigarrow s'_R[active = \texttt{true}], \Delta'} \qquad \dfrac{\dfrac{\dots}{\Omega_0, \Sigma_1, \Delta_{s_A}, \emptyset \vdash s_A \longrightarrow_{\uparrow} s'_A, \Delta'_{s_A}, \Omega'}}{s_A \in s_R.first :\ \Delta'' \rightharpoonup \Delta_{s_A}, \quad \Delta'_{s_A} \rightharpoonup \Delta_1}}{\Omega_0, \Sigma_1, \Delta_0, \emptyset \vdash s_R \longrightarrow_{\uparrow} s'_R, \Delta_1, \Omega'}$$

Activation of s_A, with omitted $s_A.active$ check.

$$\frac{\dfrac{\Delta_{s_A} \vdash s_A.active \rightsquigarrow \texttt{false}}{\Delta_{s_A} \vdash s_A \rightsquigarrow s'_A[active = \texttt{true}], \Delta'} \quad \dfrac{\cdots}{\Omega_0, \Sigma_1, \Delta' \vdash s_A.entry \rightsquigarrow s_A.entry, \Delta'_{s_A}, \Omega'}}{\Omega_0, \Sigma_1, \Delta_{s_A}, \emptyset \vdash s_A \longrightarrow_{\uparrow} s'_A, \Delta'_{s_A}, \Omega'}$$

The operation call is performed when evaluating the entry function of s_A, which just consists of the single operation call statement $f_1 \in \mathcal{SC}$:

$$\frac{\dfrac{\cdots}{\Omega^0, \Sigma_1, \Delta^0 \vdash f_1 \underset{\omega!\alpha_1}{\rightsquigarrow} f_1} \quad \dfrac{\cdots}{\Omega^0 \vdash svc_1 \overset{\omega?\alpha_1}{\hookrightarrow} svc'_1,\ \Omega^1} \qquad f_1 \in s_A.entry : \quad \Delta' \rightharpoonup \Delta^0,\ \Omega_0 \rightharpoonup \Omega^0, \quad \Omega^1 \rightharpoonup \Omega'}{\Omega_0, \Sigma_1, \Delta' \vdash s_A.entry \overset{\epsilon}{\rightsquigarrow} s_A.entry,\ \Delta'', \Omega'}$$

$$\frac{\begin{array}{l}\Sigma_1 \vdash f_1.svc \rightsquigarrow svc_1 \neq \emptyset \\ \alpha_1 = f_1.op(f_1.op) \neq \emptyset \\ \omega = channel(\Omega^0, svc_1)\end{array}}{\Omega^0, \Sigma_1, \Delta^0 \vdash f_1 \underset{\omega!\alpha_1}{\rightsquigarrow} f_1}$$

The invoked operation results in a service state transition of svc_1. Even though applying the rule Trans1 for service progress, the environment stays unchanged and $\Omega^1 = \Omega^0$, because no events are emitted when transitioning t_1 and the operation response is deferred, i.e. $\epsilon = \emptyset$. After transitioning, the current service state of svc_1 is ss_Y.

$$\frac{\begin{array}{l}t_1 \in svc_1.trans : \\ \quad t_1.src = svc_1.cur = ss_X \\ \quad \alpha_1 = t_1.operation \quad \omega = channel(\Omega^0, svc_1) \\ \quad \epsilon = t_1.respond \ \cup \ t_1.events = \emptyset \\ \quad put(\Omega^0, \epsilon) = \Omega^1 = \Omega^0\end{array}}{\Omega^0 \vdash svc_1 \overset{\omega?\alpha_1}{\hookrightarrow} svc'_1[cur = t_1.dst = ss_Y], \Omega^1}$$

In the execution step, the service svc_1 transitions with application of rule Stutter:

$$\frac{\begin{array}{l}\epsilon = svc.cur.events = \emptyset \\ put(\Omega_{svc}, \epsilon) = \Omega'_{svc} = \Omega_{svc}\end{array}}{\Omega_{svc} \vdash svc_1 \hookrightarrow svc_1,\ \Omega'_{svc}}$$

The derivation tree of this first execution step, which consists of an RCM macro and service stuttering step, clearly shows all individual evaluations and their effects on the modeled contexts. In this scenario, it can be shown, that environments Ω_0, Ω' and Ω_1 of the first execution step and Ω_i of any following execution steps $i > 1$ are equivalent. Furthermore, they do not contain operation response data required for evaluation of the *success* port condition of state s_A. The relevant parts of the derivation tree are the applied Trans and Stutter relations of the service svc_1, because they result in evaluation of the *put* function, which emits operation response or event data ϵ into an environment. However, the determined ϵ will always be the empty set until either the $op2$ or $reset$ operation get invoked, which cannot happen in the modeled scenario as the RCM instance is the only process interacting with the service.

Chapter 6

Conclusion

The EMB.RACE ecosystem with RACE-Core and RACE-Services is a novel system architecture for robot operation and programming. This thesis represents the foundation for developing the infrastructure past research prototypes. The presented formalization enables rigorous checking of model properties, proving of intended functionality and overall provides a better understanding of the employed models. This is especially important when developing the executive or other tools which interact with the model.

However, there is still room to improve the RCM semantics, as some of the currently defined SOS rules are very complex. A possible approach is to move some of the premise constraints into appropriate functions which could simplify concrete premises in the semantic rules. Futhermore, and as already stated in the beginning of Chapter 5, formal verification of the presented formalism - and instances thereof - was not conducted and is an important next step to take. In order to verify certain language properties like determinism, run-to-completion or reachability, the formal semantics have to get translated into an appropriate input language of a model checker or theorem proving system. Related to this, formal verification presented in [22, 23] for the PLEXIL coordination language might serve as a good starting point. As for the RSM, additional research and experience is surely beneficial and it has to be checked, whether all proposed functionality can be achieved with the current generic interface. Although EMB.RACE is conceived to be middleware-agnostic, it should be considered to integrate with other model-based approaches like BRICS [18], which already solves some of the faced problems.

While the formal semantics primarily address validation and verification on the way to certifiability of implemented software components, the process of formalizing already uncovered shortcomings of the syntax, leading to an overall iterative development process of the language. The ultimately resulting syntax and semantics of RCM provide very intuitive, flexible and expressive mechanisms for modeling robotic applications. The

architecture utilizes a separation of low-level (hard real-time) reflexes with handling of their occurrence in high-level task coordination to create a convenient, but safe framework for programming and operating robots. Error-handling in general can be modeled with RCM in various alternative ways, which is very valuable considering the variable system composition and dynamic environment of robotic systems. Moreover, the implemented prototypes of the core, services and clients already have proven to be very useful when realizing several demonstrators or in week-length evaluation sessions with students. Even though prototypical, the implementations also support online visualization and manipulation of the coordinaton model, which results in collaborative and rapid development of robotic tasks.

To conclude, the proposed formalisms provide a suitable basis for the development of such a sophisticated software architecture as EMB.RACE by paving the way for formal methods which are essential to ensure human safety in robotic applications.

Bibliography

[1] The mathworks: Stateflow and stateflow coder, users guide. release 13sp1 edn., 2003.

[2] Luca Aceto, Wan Fokkink, and Chris Verhoef. *Structural operational semantics.* Citeseer, 1999.

[3] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4):315–337, 1998.

[4] Alin Albu-Schäffer, Sami Haddadin, Christian Ott, Andreas Stemmer, Thomas Wimböck, and Gerd Hirzinger. The DLR lightweight robot: design and control concepts for robots in human environments. *Industrial Robot: An International Journal*, 34(5):376–385, 2007.

[5] Alin Albu-Schäffer, Christian Ott, and Gerd Hirzinger. A unified passivity-based control framework for position, torque and impedance control of flexible joint robots. *The International Journal of Robotics Research*, 26(1):23–39, 2007.

[6] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.

[7] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[8] Jean-Christophe Baillie. Urbi: Towards a universal robotic low-level programming language. In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 820–825, 2005.

[9] Vijay Baskaran, Michael Dalal, Tara Estlin, Chuck Fry, Robert Harris, Michael Iatauro, Ari Jónsson, Corina Pasareanu, and Reid Simmons. Plan execution interchance language (PLEXIL) version 1.0. *NASA Technical Memorandum*, 2007.

[10] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[11] Rainer Bischoff, Johannes Kurth, Günter Schreiber, Ralf Koeppe, Alin Albu-Schäffer, Alexander Beyer, Oliver Eiberger, Sami Haddadin, Andreas Stemmer, Gerhard Grunwald, et al. The KUKA-DLR lightweight robot arm-a new reference platform for robotics research and manufacturing. In *Robotics (ISR), 2010 41st international symposium on and 2010 6th German conference on robotics (ROBOTIK)*, pages 1–8. VDE, 2010.

[12] Jonathan Bohren and Steve Cousins. The SMACH high-level executive [ROS news]. *Robotics & Automation Magazine, IEEE*, 17(4):18–20, 2010.

[13] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide, 2/E*. Pearson Education India, 2005.

[14] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Orebäck. Orca: a component model and repository. In *Software engineering for experimental robotics*, pages 231–251. Springer, 2007.

[15] Davide Brugali and Patrizia Scandurra. Component-based robotic engineering (part i)[tutorial]. *Robotics & Automation Magazine, IEEE*, 16(4):84–96, 2009.

[16] Davide Brugali and Azamat Shakhimardanov. Component-based robotic engineering (part ii). *Robotics & Automation Magazine, IEEE*, 17(1):100–112, 2010.

[17] Herman Bruyninckx. Open robot control software: the OROCOS project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528, 2001.

[18] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The BRICS component model: a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1758–1764. ACM, 2013.

[19] Steve A Chien, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *AIPS*, pages 300–307, 2000.

[20] Alessandro De Luca, Alin Albu-Schäffer, Sami Haddadin, and Gerd Hirzinger. Collision detection and safe reaction with the DLR-III lightweight manipulator arm.

In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 1623–1630. IEEE, 2006.

[21] Gilles Dowek, César Munoz, and Corina Pasareanu. A formal analysis framework for PLEXIL. In *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*, pages 45–51, 2007.

[22] Gilles Dowek, César Munoz, and Corina Pasareanu. A small-step semantics of PLEXIL. *National Institute of Aerospace, Hampton, VA, Technical Report*, 11:2008, 2008.

[23] Gilles Dowek, César Muoz, and Camilo Rocha. Rewriting logic semantics of a plan execution language. *arXiv preprint arXiv:1002.2872*, 2010.

[24] Tobias Ende, Sami Haddadin, Sven Parusel, Tilo Wüsthoff, Marc Hassenzahl, and Alin Albu-Schäffer. A human-centered approach to robot gesture based communication within collaborative working processes. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3367–3374. IEEE, 2011.

[25] Tara Estlin, Ari Jonsson, Corina Pasareanu, Reid Simmons, Kam Tso, and Vandi Verma. Plan execution interchange language (PLEXIL). 2006.

[26] R James Firby. An investigation into reactive planning in complex domains. In *AAAI*, volume 87, pages 202–206, 1987.

[27] Lorenzo Flueckiger, Vinh To, and Hans Utz. Service-oriented robotic architecture supporting a lunar analog test. In *In International Symposium on Artificial Intelligence, Robotics, and Automation in Space (iSAIRAS*, 2008.

[28] Willow Garage. Personal robot 2 (pr2). `http://www.willowgarage.com`, February 2014.

[29] Sami Haddadin, Alin Albu-Schäffer, Alessandro De Luca, and Gerd Hirzinger. Collision detection and reaction: A contribution to safe physical human-robot interaction. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 3356–3363. IEEE, 2008.

[30] Sami Haddadin, Sven Parusel, Rico Belder, and Alin Albu-Schäffer. It is (almost) all about human safety: A novel paradigm for robot design, control, and planning. In Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaniche, editors, *Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 202–215. Springer Berlin Heidelberg, 2013.

[31] Sami Haddadin, Michael Suppa, Stefan Fuchs, Tim Bodenmüller, Alin Albu-Schäffer, and Gerd Hirzinger. Towards the robotic co-worker. In *Robotics Research*, pages 261–282. Springer, 2011.

[32] Gregoire Hamon and John Rushby. An operational semantics for stateflow. In *Fundamental Approaches to Software Engineering*, pages 229–243. Springer, 2004.

[33] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[34] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[35] Gerd Hirzinger, Norbert Sporer, Alin Albu-Schäffer, Matthias Hähnle, Rainer Krenn, A. Pascucci, and Markus Schedl. DLR's torque-controlled light weight robot III-are we reaching the technological limits now? In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 2, pages 1710–1716. IEEE, 2002.

[36] Leigh R Hochberg, Daniel Bacher, Beata Jarosiewicz, Nicolas Y Masse, John D Simeral, Joern Vogel, Sami Haddadin, Jie Liu, Sydney S Cash, Patrick van der Smagt, et al. Reach and grasp by people with tetraplegia using a neurally controlled robotic arm. *Nature*, 485(7398):372–375, 2012.

[37] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[38] Theo MV Janssen. Compositionality. 1996.

[39] Markus Klotzbücher and Herman Bruyninckx. Coordinating robotic tasks and systems with rFSM statecharts. *JOSER: Journal of Software Engineering for Robotics*, 2012.

[40] Russell Knight, Gregg Rabideau, Steve Chien, Barbara Engelhardt, and Rob Sherwood. Casper: Space exploration through continuous planning. *Intelligent Systems, IEEE*, 16(5):70–75, 2001.

[41] David Kortenkamp and Reid Simmons. *Springer Handbook of Robotics*, chapter Robotic Systems Architectures and Programming. Springer, 2008.

[42] KUKA Labs. LBR iiwa robot. `http://www.kuka-labs.com/en/service_robotics/lightweight_robotics/start.htm`, February 2014.

[43] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan, and Félix Ingrand. GenoM3: building middleware-independent robotic components. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 4627–4632, 2010.

[44] James Manyika, Michael Chui, Jacques Bughin, Richard Dobbs, Peter Bisson, and Alex Marrs. Mckinsey global institue - disruptive technologies: Advances that will transform life, business, and the global economy. `http://www.mckinsey.com/insights/business_technology/disruptive_technologies`, May 2013.

[45] Hanne Riis Nielson and Flemming Nielson. Semantics with applications: a formal introduction. 1992.

[46] Tobias Nipkow and Gerwin Klein. Concrete semantics: A proof assistant approach. `http://www21.in.tum.de/~nipkow/Concrete-Semantics`, March 2014.

[47] Lucas Bueno R Oliveira, Fernando S Osório, and Elisa Yumi Nakagawa. An investigation into the development of service-oriented robotic systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 223–228. ACM, 2013.

[48] Esben H. Ostergaard. Lightweight robot for everybody [industrial activities]. *Robotics Automation Magazine, IEEE*, 19(4):17–18, 2012.

[49] Sven Parusel, Sami Haddadin, and Alin Albu-Schäffer. Modular state-based behavior control for safe human-robot interaction: A lightweight control architecture for a lightweight robot. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4298–4305. IEEE, 2011.

[50] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer, 1994.

[51] R Peter Bonasso, R James Firby, Erann Gat, David Kortenkamp, David P Miller, and Mark G Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):237–256, 1997.

[52] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[53] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

[54] Matthias Radestock and Susan Eisenbach. Coordination in evolving systems. In *Trends in Distributed Systems CORBA and Beyond*, pages 162–176. Springer, 1996.

[55] Sekou L Remy and M Brian Blake. Distributed service-oriented robotics. *Internet Computing, IEEE*, 15(2):70–74, 2011.

[56] Rethink Robotics. Baxter robot. `http://www.rethinkrobotics.com`, February 2014.

[57] Unbounded Robotics. UBR-1 robot. `http://unboundedrobotics.com/ubr-1/specification/`, February 2014.

[58] Christian Schlegel, Andreas Steck, and Alexander Lotz. *Robotic Systems - Applications, Control and Programming*, chapter Robotic Software Systems: From Code-Driven to Model-Driven Software Development. InTech, 2012.

[59] Christian Schlegel and Robert Worz. The software framework smartsoft for implementing sensorimotor systems. In *Intelligent Robots and Systems, 1999. IROS'99. Proceedings. 1999 IEEE/RSJ International Conference on*, volume 3, pages 1610–1616. IEEE, 1999.

[60] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer, 2008.

[61] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, pages 1931–1937, 1998.

[62] Vandi Verma, Tara Estlin, Ari Jonsson, Corina Pasareanu, Reid Simmons, and Kam Tso. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *Proc. of Intl. Symp. on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS), Munich, Germany*, 2005.

[63] Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. The CLARAty architecture for robotic autonomy. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 1, pages 1–121, 2001.

[64] Michael von der Beeck. A comparison of statecharts variants. In *Formal techniques in real-time and fault-tolerant systems*, pages 128–148, 1994.

[65] Michael von der Beeck. A structured operational semantics for uml-statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002.

Appendix A

Demonstrator

In course of this thesis a variety of demonstrative applications were developed using EMB.RACE to validate and also improve different aspects of the system. This chapter describes one of these deomstrators, which integrates many different services to achieve the very common task of autonomously picking and placing objects.

The setup consists of an LWR-III robotic manipulator, equipped with a Schunk WSG50 gripper, on which a smartphone serving as graphical display and input device is mounted. Additionally, a Kinect depth camera is positioned in the scene to detect objets on a table. The whole system is depicted in Figure A.1.

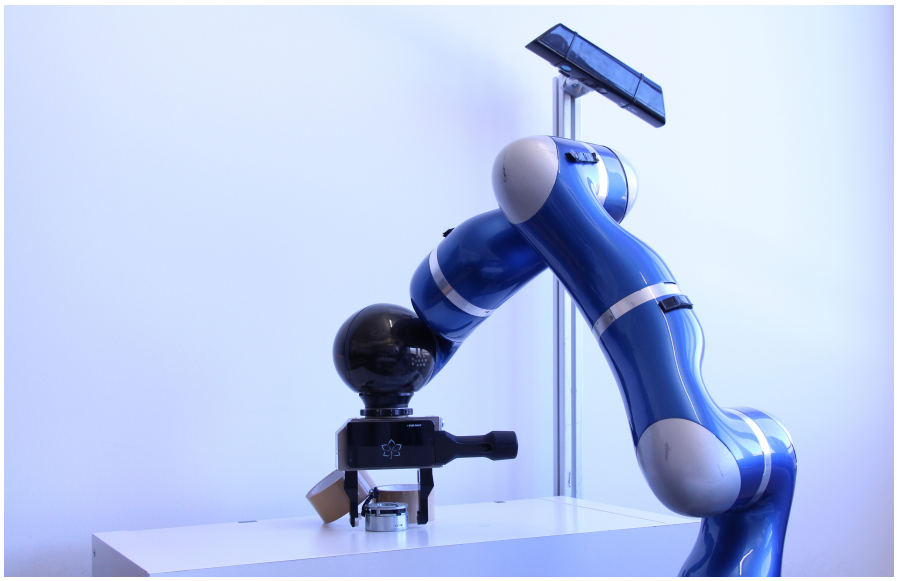


Figure A.1: Demonstrator setup with LWR-III, gripper, display and the kinect depth camera

As the camera is arbitrarily mounted in the scenario, it has to be spatially registered relative to the robot. This registration is also called extrinsic calibration, and is the first task implemented in the demonstrator. After calibration was successful, the system can recognize objects and calculate their poses. This leads to the second and main task of the example scenario, which is, autonomously picking detected objects and placing them somewhere else.

To conveniently perform both, calibration and pick-place, a variety of services are employed in this scenario:

- **Robot.** The robot service is used to control the robot's movements and specify its safety parameters and reflex behavior.
- **Gripper.** A service interface to control linear gripper movements like grasping or releasing with certain width, velocity or force.
- **KinectDriver.** This component retrieves color and depth images from the Kinect, calculates the point cloud and provides the data to other services.
- **ObjectRecognition.** Continuously recognizes objects in a given point cloud and reports detected objects with pose and confidence as events.
- **Database.** A service providing persistent storage for all kinds of data. In this case, it is used to store poses and transforms in between task executions.
- **Display.** A service used to show instructions and provide user interfaces to the human operator. In this demonstrator, the smartphone mounted on the robot is interfaced by this service.
- **Audio.** Provides operations to play sounds for user feedback purposes.

All these services are coordinated by the RCM states implementing the tasks. At the top-level, both tasks are realized by four different composite states, namely *CameraCalibration*, *SearchObject*, *Pick* and *Place*, where most of the implemented functionality is outlined in Section 5.1.

CameraCalibration is done by letting the operator move the robot into the field of view of the camera and detecting the robot's end-effector as object with the appropriate service. By averaging multiple measurements, the relative transformation from camera to robot base is calculated and stored in the database. The display and audio service are used to instruct to user and give feedback during the calibration process.

Place retrieves, similar to *Pick*, a parameterized pose from the database service. Should the requested pose be unavailable, a fallback on teaching the pose via gravity compensation is performed. In the nominal case, the robot approaches, places and retreats robustly with integrated handling of potential collisions.

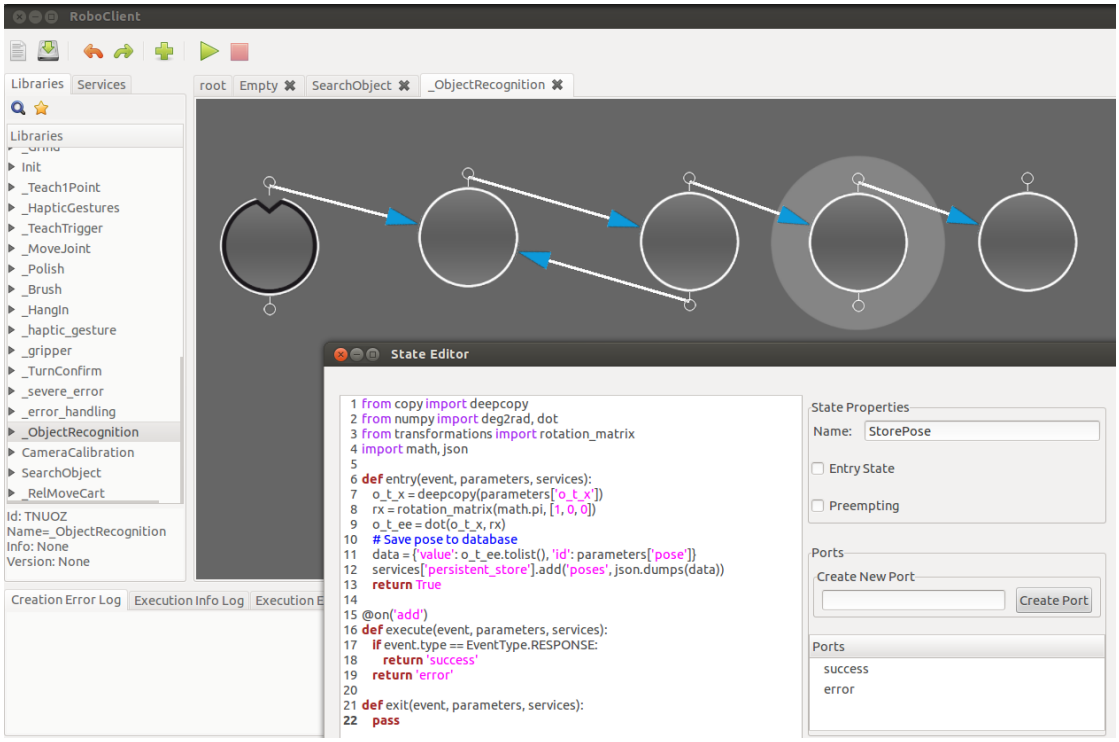


Figure A.2: RACE-Pro client visualizing the composite *ObjectRecognition* state, while detailed functionality and port configuration of the *StorePose* state are displayed in the state editor window.

The application is exemplarily visualized by the two clients RACE-Pro and RACE-One in Figures A.2 and A.3 respectively. The former shows the sequential structure of the composite *ObjectRecognition* state, while also allowing for sophisticated configuration of a state's functions, ports or parameters via the state editor window. Note that the depicted screenshot is from an early prototype, which uses a slightly different model syntax. With a comprehensive user interface for all RCM features, the RACE-Pro client addresses experienced users and task developers. RACE-One, on the other hand, maps the full RCM to two layers: skills and tasks. This abstraction is used to translate the complex hierarchical structure of RCM to the simpler composition of skills in tasks. Both still represent RCM states, but limit complexity as also the sequential composition is restricted to a single nominal flow. Besides assembling into tasks, skills can be parameterized by a context menu. RACE-One is developed for novice users to create or change robot applications in an intuitive yet flexible manner .

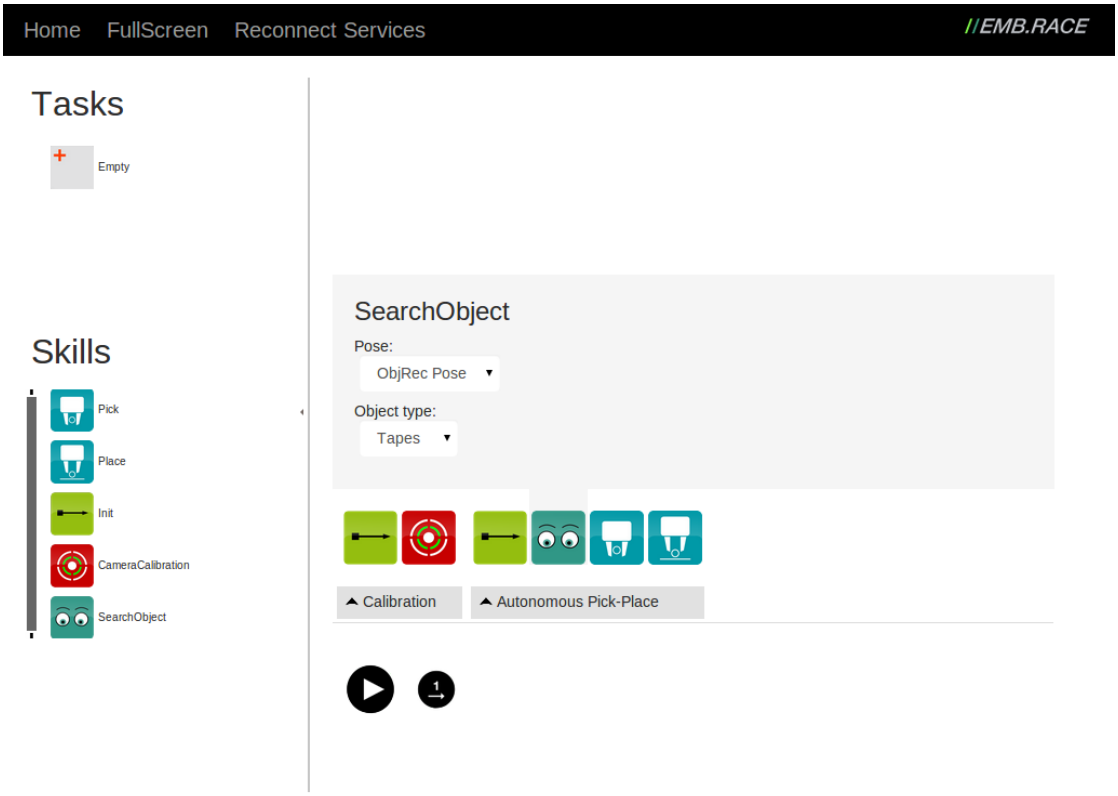


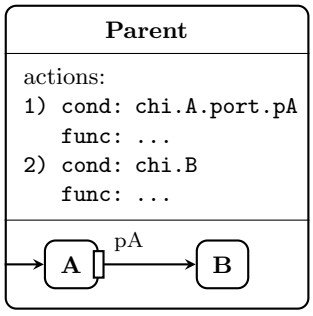
Figure A.3: RACE-One client depicting the two tasks of the demonstrative application and highlighting the context menu of state *SearchObject*, which allows to change state parameters.

Appendix B

Derivation trees

This appendix includes formal derivation trees of the examples considered in Chapter 5. For every appended derivation, the graphical representation and scenario description is also included here. Note that some premises, like checking whether states are active, are omitted and $[\neg active]$ is used as shorthand notation for $[active = \texttt{false}]$ due to the limited space .

Action execution sequence



Scenario:

- Port p_A gets activated in macro step 1 and is connected to state B
- Two actions a_1 and a_2 react on activation of p_A and B respectively
- In what sequence are these actions executed relative to the entry function of B?

Formal derivation:
When assuming that composite state s_p and the first state s_A are active when evaluating macro step 1, the execution of this scenario is described by a single Exec/Macro step.

Execution: One Exec/Macro step

$$\frac{\dfrac{\dots}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1), \Omega_1}}{(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1, \Delta_1)}$$

Macro 1: In this scenario, the only macro step consists of three micro steps

$$\frac{\Sigma_1 = \Omega_0 \qquad \dfrac{\dfrac{\dots}{\Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta'} \quad \dfrac{\dots}{\Sigma_1 \vdash \Delta' \Longrightarrow \Delta'''} \quad \dfrac{\dots}{\Sigma_1 \vdash \Delta''' \Longrightarrow \Delta_1}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow_{\downarrow} \Delta_1}}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1), \Omega_1}$$

Micro 1-1: In the first micro step, SProp is applicable to s_P where no action conditions are fulfilled.

$$\frac{\dfrac{\dfrac{\Sigma_1, \Delta^0 \vdash a_1.cond \rightsquigarrow \texttt{false}}{\Sigma_1, \Delta^0 \vdash a_1 \rightsquigarrow a_1} \quad \dfrac{\Sigma_1, \Delta^0 \vdash a_2.cond \rightsquigarrow \texttt{false}}{\Sigma_1, \Delta^0 \vdash a_2 \rightsquigarrow a_2} \quad \dfrac{\dots}{\Sigma_1, \Delta_{s_A} \vdash s_A \longrightarrow s'_A, \Delta'''_{s_A}}}{\forall a_i \in s_P.actions: \ i = 1...n, \ \Delta_0 \rightharpoonup \Delta^0 \qquad s_A \in active\_states(\Delta_0, s_P): \ \Delta_0 \rightharpoonup \Delta_{s_A}, \ \Delta'''_{s_A} \rightharpoonup \Delta'}}{\Sigma_1, \Delta_0 \vdash s_P \longrightarrow s_P, \Delta'}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta'}$$

PTrigger of s_A, as p_A activates in this macro step by definition of the scenario.

$$\frac{\Delta_{s_A} \vdash s_A.active \rightsquigarrow \texttt{true} \qquad \dfrac{\Sigma_1, \Delta_{s_A} \vdash p_A.cond \rightsquigarrow \texttt{true} \quad \dfrac{\dots}{\Sigma_3, \Delta_{s_A} \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta''_{s_A}} \quad \Delta''_{s_A} \vdash p_A \rightsquigarrow p'_A[active = \texttt{true}], \Delta'''_{s_A}}{p_A \in s_A.ports:}}{\Sigma_1, \Delta_{s_A} \vdash s_A \longrightarrow s'_A, \Delta'''_{s_A}}$$

Deactivation of s_A:

$$\frac{\Delta_{s_A} \vdash s_A.active \rightsquigarrow \texttt{true} \quad \Sigma_1, \Delta_{s_A} \vdash s_A.exit \rightsquigarrow s_A.exit, \Delta'_{s_A} \quad \Sigma_1, \Delta'_{s_A} \vdash s_A \rightsquigarrow s'_A[active = \texttt{false}], \Delta''_{s_A}}{\Sigma_1, \Delta_{s_A} \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta''_{s_A}}$$

Micro 1-2: PTrans of s_P with active port p_A, where $(\Delta'_{p_A}, \Delta''_{s_B}) \rightharpoonup \Delta'''$

$$\dfrac{\dfrac{\Sigma_1 \vdash executed(\Sigma_1, a_1) \rightsquigarrow \texttt{false}}{\Sigma_1, \Delta^0 \vdash a_1.cond \rightsquigarrow \texttt{true}} \quad \dfrac{\Sigma_1, \Delta^0 \vdash a_1.func \rightsquigarrow a_1.func, \Delta^1}{\Sigma_1, \Delta^0 \vdash a_1 \rightsquigarrow a_1, \Delta^1} \quad \dfrac{\Sigma_1, \Delta^1 \vdash a_2.cond \rightsquigarrow \texttt{false}}{\Sigma_1, \Delta^1 \vdash a_2 \rightsquigarrow a_2} \quad \dfrac{\cdots}{\Sigma_1, \Delta_{s_B}, p_A \vdash s_B \longrightarrow_{\uparrow} s'_B, \Delta''_{s_B}} \quad \Delta_{p_A} \vdash p_A \rightsquigarrow p'_A[\neg active], \Delta'_{p_A}}{\dfrac{\forall a_i \in s_P.actions : \ i = 1...n, \ \Delta' \rightharpoonup \Delta^0, \ \Delta^1 \rightharpoonup \Delta'' \qquad (p_A, s_B) \in active\_ports(\Delta', s_P) : \ \Delta'' \rightharpoonup (\Delta_{p_A}, \Delta_{s_B})}{\Sigma_1, \Delta' \vdash s_P \longrightarrow s_P, \Delta'''}}{\Sigma_1 \vdash \Delta' \Longrightarrow \Delta'''}$$

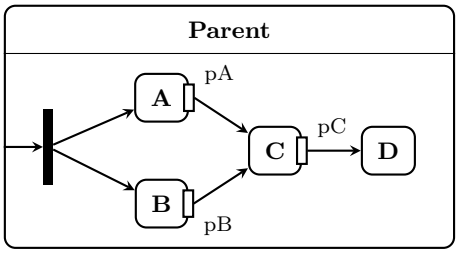
Activation of s_B (from port p_A):

$$\dfrac{\Delta_{s_B} \vdash s_B.active \rightsquigarrow \texttt{false} \quad \Delta_{s_B} \vdash s_B \rightsquigarrow s'_B[active = \texttt{true}], \Delta'_{s_B} \quad \Sigma_1, \Delta'_{s_B} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_B}}{\Sigma_2, \Delta_{s_B}, p_A \vdash s_B \longrightarrow_{\uparrow} s'_B, \Delta''_{s_B}}$$

Micro 1-3: SProp of s_P, with action execution as the condition of a_2 is fulfilled. s_B is the only active state, but does not specify any actions or children to evaluate.

$$\dfrac{\dfrac{\dfrac{\Sigma_1 \vdash executed(\Sigma_1, a_2) \rightsquigarrow \texttt{false}}{\Sigma_1, \Delta^0 \vdash a_2.cond \rightsquigarrow \texttt{true}} \quad \dfrac{\Sigma_1, \Delta^0 \vdash a_1.cond \rightsquigarrow \texttt{false}}{\Sigma_1, \Delta^0 \vdash a_1 \rightsquigarrow a_1} \quad \dfrac{\Sigma_1, \Delta^0 \vdash a_2.func \rightsquigarrow a_2.func, \Delta^1}{\Sigma_1, \Delta^0 \vdash a_2 \rightsquigarrow a_2, \Delta^1} \quad \dfrac{\Delta_{s_B} \vdash s_B.active \rightsquigarrow \texttt{true} \quad s_B.actions = \emptyset \quad s_B.children = \emptyset}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow s_B}}{\dfrac{\forall a_i \in s_P.actions : \ i = 1...n, \ \Delta''' \rightharpoonup \Delta^0, \ \Delta^1 \rightharpoonup \Delta_1 \qquad s_B \in active\_states(\Delta_1, s_P) : \ \Delta_1 \rightharpoonup \Delta_{s_B}}{\Sigma_1, \Delta''' \vdash s_P \longrightarrow s_P, \Delta_1}}{\Sigma_1 \vdash \Delta''' \Longrightarrow \Delta_1}$$

Concurrent port activation (same destination)



Scenario:

- Ports p_A and p_B activate both in macro step 2
- The condition of p_C is fulfilled in macro step 3

Formal derivation:

Execution consists of three different steps:

$$(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1, \Delta_1) \mapsto (\Omega_2, \Sigma_2, \Delta_2) \mapsto (\Omega_3, \Sigma_3, \Delta_3)$$

Where each is described by a corresponding macro step:

$$\frac{\frac{\dots}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1), \Omega_1}}{(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1, \Delta_1)}$$

$$\frac{\frac{\dots}{\Omega_1 \vdash (\Sigma_1, \Delta_1) \ast\!\!\longrightarrow (\Sigma_2, \Delta_2), \Omega_2}}{(\Omega_1, \Sigma_1, \Delta_1) \mapsto (\Omega_2, \Sigma_2, \Delta_2)}$$

$$\frac{\frac{\dots}{\Omega_2 \vdash (\Sigma_2, \Delta_2) \ast\!\!\longrightarrow (\Sigma_3, \Delta_3), \Omega_3}}{(\Omega_2, \Sigma_2, \Delta_2) \mapsto (\Omega_3, \Sigma_3, \Delta_3)}$$

In the following, environments Ω are omitted, as no services are involved in this scenario.

Macro 1: The first micro step is assumed to activate s_P, where the second micro step transitions the barrier

$$\dfrac{\Sigma_1 = \Omega_0 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Sigma_1, \Delta_0, \emptyset \vdash s_P \longrightarrow_\uparrow s'_P, \Delta'''}}{\Sigma_1, \Delta_0 \vdash s_P \longrightarrow s'_P, \Delta'''}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta'''} \qquad \dfrac{\dots}{\Sigma_1 \vdash \Delta''' \Longrightarrow \Delta_1}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow_\downarrow \Delta_1}}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1)}$$

Micro 1-1: Activation of s_P, where the $s_P.active$ premise is omitted. Application results in activation of first element b via Rule BAct1

$$\dfrac{\Delta_0 \vdash s_P \rightsquigarrow s'_P[active = \texttt{true}], \Delta' \qquad \Sigma_1, \Delta' \vdash s_P.entry \rightsquigarrow s_P.entry, \Delta'' \qquad \dfrac{\Delta_b \vdash b.active \rightsquigarrow \texttt{false} \quad \Delta_b \vdash b \rightsquigarrow b'[active = \texttt{true}], \Delta'_b}{\Sigma_1, \Delta_b, \emptyset \vdash b \longrightarrow_\uparrow b', \Delta'_b} \qquad b \in s_P.first : \ \Delta'' \multimap \Delta_b, \quad \Delta'_b \rightharpoonup \Delta'''}{\Sigma_1, \Delta_0, \emptyset \vdash s_P \longrightarrow_\uparrow s'_P, \Delta'''}$$

Micro 1-2: Transitioning of barrier b via application of rule PTrans2 on s_P. Parameter resolution and active check of s_P is omitted for the sake of breviety. Also context propagation is done by $\Delta''' \multimap \Delta_b$ and $(\Delta'_{s_A} \Delta'_{s_B}, \Delta'_b) \rightharpoonup \Delta_1$.

$$\dfrac{\dfrac{b \in active\_barriers(\Delta''', s_P) \qquad \dfrac{\dots}{\Sigma_1, \Delta_{s_A}, \emptyset \vdash s_A \longrightarrow_\uparrow s'_A, \Delta''_{s_A}} \quad s_A \in b.out : \ \Delta_b \multimap \Delta_{s_A} \qquad \dfrac{\dots}{\Sigma_1, \Delta_{s_B}, \emptyset \vdash s_B \longrightarrow_\uparrow s'_B, \Delta''_{s_B}} \quad s_B \in b.out : \ \Delta_b \multimap \Delta_{s_B} \qquad \Delta_b \vdash b \rightsquigarrow b'[\neg active], \Delta'_b}{\Sigma_1, \Delta''' \vdash s_P \longrightarrow s_P, \Delta_1}}{\Sigma_1 \vdash \Delta''' \Longrightarrow \Delta_1}$$

Activation of s_A:

$$\frac{\Delta_{s_A} \vdash s_A.active \rightsquigarrow \texttt{false} \quad \Delta_{s_A} \vdash s_A \rightsquigarrow s'_A[active = \texttt{true}], \Delta'_{s_A} \quad \Sigma_1, \Delta'_{s_A} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_A}}{\Sigma_1, \Delta_{s_A}, \emptyset \vdash s_A \longrightarrow_{\uparrow} s'_A, \Delta''_{s_A}}$$

Activation of s_B:

$$\frac{\Delta_{s_B} \vdash s_B.active \rightsquigarrow \texttt{false} \quad \Delta_{s_B} \vdash s_B \rightsquigarrow s'_B[active = \texttt{true}], \Delta'_{s_B} \quad \Sigma_1, \Delta'_{s_B} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_B}}{\Sigma_1, \Delta_{s_B}, \emptyset \vdash s_B \longrightarrow_{\uparrow} s'_B, \Delta''_{s_B}}$$

Macro 2: Port conditions of p_A and p_B are fulfilled by scenario definition, which leads to activation of state s_C. The macro step consists of two micro steps:

$$\frac{\Sigma_2 = \Omega_1 \qquad \dfrac{\dfrac{\dots}{\Sigma_2 \vdash \Delta_1 \Longrightarrow \Delta'} \quad \dfrac{\dots}{\Sigma_2 \vdash \Delta' \Longrightarrow \Delta_2}}{\Sigma_2 \vdash \Delta_1 \Longrightarrow_{\downarrow} \Delta_2}}{\Omega_1 \vdash (\Sigma_1, \Delta_1) \ast\!\!\longrightarrow (\Sigma_2, \Delta_2)}$$

Micro 2-1: Application of SProp for s_P and PTrigger for states s_A and s_B. The changed contexts propagate up by $(\Delta'''_{s_A}, \Delta'''_{s_B}) \rightharpoonup \Delta'$.

$$\frac{\dfrac{\Delta_1 \vdash s_P.active \rightsquigarrow \texttt{true} \quad s_P.actions = \emptyset \quad s_A \in active\_states(\Delta_1, s_P) : \ \Delta_1 \rightharpoonup \Delta_{s_A} \quad s_B \in active\_states(\Delta_1, s_P) : \ \Delta_1 \rightharpoonup \Delta_{s_B}}{\Sigma_2, \Delta_1 \vdash s_P \longrightarrow s_P, \Delta'} \qquad \dfrac{\dots}{\Sigma_2, \Delta_{s_A} \vdash s_A \longrightarrow s'_A, \Delta'''_{s_A}} \qquad \dfrac{\dots}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow s'_B, \Delta'''_{s_B}}}{\Sigma_2 \vdash \Delta_1 \Longrightarrow \Delta'}$$

PTrigger of s_A for port p_A:

$$\frac{\Delta \vdash s_A.active \rightsquigarrow \texttt{true} \quad p_A \in s_A.ports : \quad \Sigma_2, \Delta_{s_A} \vdash p_A.cond \rightsquigarrow \texttt{true} \quad \dfrac{\dots}{\Sigma_2, \Delta_{s_A} \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta''_{s_A}} \quad \Delta''_{s_A} \vdash p_A \rightsquigarrow p'_A[active = \texttt{true}], \Delta'''_{s_A}}{\Sigma_2, \Delta_{s_A} \vdash s_A \longrightarrow s'_A, \Delta'''_{s_A}}$$

Deactivation of s_A:

$$\frac{\Delta_{s_A} \vdash s_A.active \rightsquigarrow \texttt{true} \quad \Sigma_2, \Delta_{s_A} \vdash s_A.exit \rightsquigarrow s_A.exit, \Delta'_{s_A} \quad \Sigma_2, \Delta'_{s_A} \vdash s_A \rightsquigarrow s'_A[active = \texttt{false}], \Delta''_{s_A}}{\Sigma_2, \Delta_{s_A} \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta''_{s_A}}$$

PTrigger of s_B for port p_B:

$$\frac{\Delta \vdash s_B.active \rightsquigarrow \texttt{true} \quad p_B \in s_B.ports : \quad \Sigma_2, \Delta_{s_B} \vdash p_B.cond \rightsquigarrow \texttt{true} \quad \dfrac{\dots}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow_{\downarrow} s'_B, \Delta''_{s_B}} \quad \Delta''_{s_B} \vdash p_B \rightsquigarrow p'_B[active = \texttt{true}], \Delta'''_{s_B}}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow s'_B, \Delta'''_{s_B}}$$

Deactivation of s_B:

$$\frac{\Delta_{s_B} \vdash s_B.active \rightsquigarrow \texttt{true} \quad s_B.children = \emptyset \quad \Sigma_2, \Delta_{s_B} \vdash s_B.exit \rightsquigarrow s_B.exit, \Delta'_{s_B} \quad \Sigma_2, \Delta'_{s_B} \vdash s_B \rightsquigarrow s'_B[active = \texttt{false}], \Delta''_{s_B}}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow_{\downarrow} s'_B, \Delta''_{s_B}}$$

Micro 2-2: Application of PTrans1 on s_P with two active ports p_A and p_B, where evaluation order is not explicitly defined and this derivation evaluates p_A first. After evaluating all transitions, the contexts are propagated up with $(\Delta''_{s_C}, \Delta'_{p_A}, \Delta'_{p_B}) \rightharpoonup \Delta_2$. Note that the $s_P.active$ and action related premises are omitted.

$$\frac{\dfrac{\dfrac{\dots}{\Sigma_2, \Delta_{s_C}, p_A \vdash s_C \longrightarrow_{\uparrow} s'_C, \Delta''_{s_C}} \quad \Delta_{p_A} \vdash p_A \rightsquigarrow p'_A[\neg active], \Delta'_{p_A} \quad \dfrac{\Delta_{s_C} \vdash s_C.active \rightsquigarrow \texttt{true}}{\Sigma_2, \Delta_{s_C}, p_B \vdash s_C \longrightarrow_{\uparrow} s_C} \quad \Delta_{p_B} \vdash p_B \rightsquigarrow p'_B[\neg active], \Delta'_{p_B}}{(p_A, s_C) \in active\_ports(\Delta', s_P) : \ \Delta' \multimap (\Delta_{p_A}, \Delta_{s_C}) \qquad (p_B, s_C) \in active\_ports(\Delta', s_P) : \ \Delta' \multimap (\Delta_{p_B}, \Delta_{s_C})}}{\Sigma_2, \Delta' \vdash s_P \longrightarrow s_P, \Delta_2}}{\Sigma_2 \vdash \Delta' \Longrightarrow \Delta_2}$$

Activation of s_C (from port p_A):

$$\frac{\Delta_{s_C} \vdash s_C.active \rightsquigarrow \texttt{false} \quad \Delta_{s_C} \vdash s_C \rightsquigarrow s'_C[active = \texttt{true}], \Delta'_{s_C} \quad \Sigma_1, \Delta'_{s_C} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_C}}{\Sigma_2, \Delta_{s_C}, p_A \vdash s_C \longrightarrow_{\uparrow} s'_C, \Delta''_{s_C}}$$

Macro 3: As port p_C has its condition fulfilled in this macro step, application of SProp and PTrigger constitute the first micro step, where a transition via rule PTrans1 represents the second micro step, before reaching quiescence

$$\frac{\Sigma_3 = \Omega_2 \qquad \dfrac{\dfrac{\dots}{\Sigma_3 \vdash \Delta_2 \Longrightarrow \Delta'} \quad \dfrac{\dots}{\Sigma_3 \vdash \Delta' \Longrightarrow \Delta_3}}{\Sigma_3 \vdash \Delta_2 \Longrightarrow_{\downarrow} \Delta_3}}{\Omega_2 \vdash (\Sigma_2, \Delta_2) \ast\!\!\longrightarrow (\Sigma_3, \Delta_3)}$$

Micro 3-1: SProp of s_P with PTrigger of only active state s_C

$$\frac{\Delta_2 \vdash s_P.active \rightsquigarrow \texttt{true} \quad s_P.actions = \emptyset \quad s_C \in active\_states(\Delta_2, s_P) : \ \Delta_2 \multimap \Delta_{s_C} \quad \Delta'''_{s_C} \rightharpoonup \Delta' \qquad \dfrac{\dots}{\Sigma_3, \Delta_{s_C} \vdash s_C \longrightarrow s'_C, \Delta'''_{s_C}}}{\Sigma_3, \Delta_2 \vdash s_P \longrightarrow s_P, \Delta'}}{\Sigma_3 \vdash \Delta_2 \Longrightarrow \Delta'}$$

PTrigger of s_C for port p_C:

$$\frac{\Delta \vdash s_C.active \rightsquigarrow \texttt{true} \quad p_C \in s_C.ports : \quad \Sigma_3, \Delta_{s_C} \vdash p_C.cond \rightsquigarrow \texttt{true} \quad \dfrac{\dots}{\Sigma_3, \Delta_{s_C} \vdash s_C \longrightarrow_{\downarrow} s'_C, \Delta''_{s_C}} \quad \Delta''_{s_C} \vdash p_C \rightsquigarrow p'_C[active = \texttt{true}], \Delta'''_{s_C}}{\Sigma_3, \Delta_{s_C} \vdash s_C \longrightarrow s'_C, \Delta'''_{s_C}}$$

Deactivation of s_C:

$$\frac{\Delta_{s_C} \vdash s_C.active \rightsquigarrow \texttt{true} \quad s_C.children = \emptyset \quad \Sigma_3, \Delta_{s_C} \vdash s_C.exit \rightsquigarrow s_C.exit, \Delta'_{s_C} \quad \Sigma_3, \Delta'_{s_C} \vdash s_C \rightsquigarrow s'_C[active = \texttt{false}], \Delta''_{s_C}}{\Sigma_3, \Delta_{s_C} \vdash s_C \longrightarrow_{\downarrow} s'_C, \Delta''_{s_C}}$$

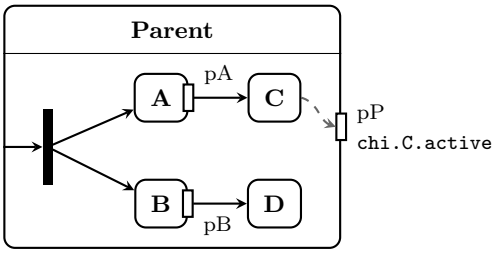
Micro 3-2: PTrans of s_P with active port p_C, where the active check of s_P is omitted again.

$$\frac{\dfrac{s_P.actions = \emptyset \quad (p_C, s_D) \in active\_ports(\Delta', s_P) : \ \Delta' \multimap (\Delta_{p_C}, \Delta_{s_D}) \qquad (\Delta'_{p_C}, \Delta''_{s_D}) \rightharpoonup \Delta_3}{\Sigma_3, \Delta' \vdash s_P \longrightarrow s_P, \Delta_3}}{\Sigma_3 \vdash \Delta' \Longrightarrow \Delta_3} \quad \frac{\dots}{\Sigma_3, \Delta_{s_D}, p_C \vdash s_D \longrightarrow_{\uparrow} s'_D, \Delta''_{s_D}} \quad \Delta_{p_C} \vdash p_C \rightsquigarrow p'_C[active = \texttt{false}], \Delta'_{p_C}$$

Activation of s_D (from port p_C):

$$\frac{\Delta_{s_D} \vdash s_D.active \rightsquigarrow \texttt{false} \quad \Delta_{s_D} \vdash s_D \rightsquigarrow s'_D[active = \texttt{true}], \Delta'_{s_D} \quad \Sigma_1, \Delta'_{s_D} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_D}}{\Sigma_2, \Delta_{s_D}, p_C \vdash s_D \longrightarrow_{\uparrow} s'_D, \Delta''_{s_D}}$$

Concurrent port activation (different destinations)



Scenario:

- Ports p_A and p_B activate both in macro step 2
- Parental port p_P triggers on activation of state C
- Will D get entered before p_P preempts all active states in Parent? Does it depend on evaluation order of transitions from p_A and p_B?

Formal derivation:

Execution consists of two different steps:

$$(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1, \Delta_1) \mapsto (\Omega_2, \Sigma_2, \Delta_2)$$

Where each is described by a corresponding macro step:

$$\frac{\dfrac{\dots}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1), \Omega_1}}{(\Omega_0, \Sigma_0, \Delta_0) \mapsto (\Omega_1, \Sigma_1, \Delta_1)}$$

$$\frac{\dfrac{\dots}{\Omega_1 \vdash (\Sigma_1, \Delta_1) \ast\!\!\longrightarrow (\Sigma_2, \Delta_2), \Omega_2}}{(\Omega_1, \Sigma_1, \Delta_1) \mapsto (\Omega_2, \Sigma_2, \Delta_2)}$$

Macro 1: The first micro step is assumed to activate s_P, where the second micro step transitions the barrier

$$\frac{\Sigma_1 = \Omega_0 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Sigma_1, \Delta_0, \emptyset \vdash s_P \longrightarrow_{\uparrow} s'_P, \Delta'''}}{\Sigma_1, \Delta_0 \vdash s_P \longrightarrow s'_P, \Delta'''}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow \Delta'''} \qquad \dfrac{\dots}{\Sigma_1 \vdash \Delta''' \Longrightarrow \Delta_1}}{\Sigma_1 \vdash \Delta_0 \Longrightarrow_{\downarrow} \Delta_1}}{\Omega_0 \vdash (\Sigma_0, \Delta_0) \ast\!\!\longrightarrow (\Sigma_1, \Delta_1)}$$

Micro 1-1: Activation of s_P, where the $s_P.active$ premise is omitted. Application results in activation of first element b via Rule BAct1

$$\frac{\Delta_0 \vdash s_P \rightsquigarrow s'_P[active = \texttt{true}], \Delta' \quad \Sigma_1, \Delta' \vdash s_P.entry \rightsquigarrow s_P.entry, \Delta'' \quad b \in s_P.first : \ \Delta'' \rightharpoonup \Delta_b, \quad \Delta'_b \rightharpoonup \Delta'''}{\Sigma_1, \Delta_0, \emptyset \vdash s_P \longrightarrow_\uparrow s'_P, \Delta'''}$$

$$\frac{\Delta_b \vdash b.active \rightsquigarrow \texttt{false} \quad \Delta_b \vdash b \rightsquigarrow b'[active = \texttt{true}], \Delta'_b}{\Sigma_1, \Delta_b, \emptyset \vdash b \longrightarrow_\uparrow b', \Delta'_b}$$

Micro 1-2: Transitioning of barrier b via application of rule PTrans2 on s_P. Parameter resolution and active check of s_P is omitted for the sake of breviety. Also context propagation is done by $\Delta''' \rightharpoonup \Delta_b$ and $(\Delta'_{s_A} \Delta'_{s_B}, \Delta'_b) \rightharpoonup \Delta_1$.

$$\frac{\frac{b \in active\_barriers(\Delta''', s_P) \quad s_A \in b.out : \ \Delta_b \rightharpoonup \Delta_{s_A} \quad \frac{\dots}{\Sigma_1, \Delta_{s_A}, \emptyset \vdash s_A \longrightarrow_\uparrow s'_A, \Delta''_{s_A}} \quad s_B \in b.out : \ \Delta_b \rightharpoonup \Delta_{s_B} \quad \frac{\dots}{\Sigma_1, \Delta_{s_B}, \emptyset \vdash s_B \longrightarrow_\uparrow s'_B, \Delta''_{s_B}} \quad \Delta_b \vdash b \rightsquigarrow b'[\neg active], \Delta'_b}{\Sigma_1, \Delta''' \vdash s_P \longrightarrow s_P, \Delta_1}}{\Sigma_1 \vdash \Delta''' \Longrightarrow \Delta_1}$$

Activation of s_A:

$$\frac{\Delta_{s_A} \vdash s_A.active \rightsquigarrow \texttt{false} \quad \Delta_{s_A} \vdash s_A \rightsquigarrow s'_A[active = \texttt{true}], \Delta'_{s_A} \quad \Sigma_1, \Delta'_{s_A} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_A}}{\Sigma_1, \Delta_{s_A}, \emptyset \vdash s_A \longrightarrow_\uparrow s'_A, \Delta''_{s_A}}$$

Activation of s_B:

$$\frac{\Delta_{s_B} \vdash s_B.active \rightsquigarrow \texttt{false} \quad \Delta_{s_B} \vdash s_B \rightsquigarrow s'_B[active = \texttt{true}], \Delta'_{s_B} \quad \Sigma_1, \Delta'_{s_B} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_B}}{\Sigma_1, \Delta_{s_B}, \emptyset \vdash s_B \longrightarrow_\uparrow s'_B, \Delta''_{s_B}}$$

Macro 2: Port conditions of p_A and p_B are fulfilled by scenario definition, which leads to activation of state s_C and s_D respectively. The macro step then consists of three micro steps:

$$\frac{\Sigma_2 = \Omega_1 \qquad \dfrac{\overset{\dots}{\Sigma_2 \vdash \Delta_1 \Longrightarrow \Delta'} \quad \overset{\dots}{\Sigma_2 \vdash \Delta' \Longrightarrow \Delta''} \quad \overset{\dots}{\Sigma_2 \vdash \Delta'' \Longrightarrow \Delta_2}}{\Sigma_2 \vdash \Delta_1 \Longrightarrow_{\downarrow} \Delta_2}}{\Omega_1 \vdash (\Sigma_1, \Delta_1) \ast\!\longrightarrow (\Sigma_2, \Delta_2)}$$

Micro 2-1: Application of SProp for s_P and PTrigger for states s_A and s_B. The changed contexts propagate up by $(\Delta'''_{s_A}, \Delta'''_{s_B}) \rightharpoonup \Delta'$.

$$\frac{\dfrac{\Delta_1 \vdash s_P.active \rightsquigarrow \texttt{true} \quad s_P.actions = \emptyset \quad s_A \in active\_states(\Delta_1, s_P) : \ \Delta_1 \rightharpoonup \Delta_{s_A} \quad s_B \in active\_states(\Delta_1, s_P) : \ \Delta_1 \rightharpoonup \Delta_{s_B}}{\Sigma_2, \Delta_1 \vdash s_P \longrightarrow s_P, \Delta'} \qquad \dfrac{\overset{\dots}{\Sigma_2, \Delta_{s_A} \vdash s_A \longrightarrow s'_A, \Delta'''_{s_A}} \qquad \overset{\dots}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow s'_B, \Delta'''_{s_B}}}{\Sigma_2 \vdash \Delta_1 \Longrightarrow \Delta'}$$

PTrigger of s_A for port p_A:

$$\frac{\dfrac{\Delta \vdash s_A.active \rightsquigarrow \texttt{true} \quad p_A \in s_A.ports : \qquad \Sigma_2, \Delta_{s_A} \vdash p_A.cond \rightsquigarrow \texttt{true} \quad \overset{\dots}{\Sigma_2, \Delta_{s_A} \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta''_{s_A}} \quad \Delta''_{s_A} \vdash p_A \rightsquigarrow p'_A[active = \texttt{true}], \Delta'''_{s_A}}{\Sigma_2, \Delta_{s_A} \vdash s_A \longrightarrow s'_A, \Delta'''_{s_A}}}{}$$

Deactivation of s_A:

$$\frac{\Delta_{s_A} \vdash s_A.active \rightsquigarrow \texttt{true} \quad \Sigma_2, \Delta_{s_A} \vdash s_A.exit \rightsquigarrow s_A.exit, \Delta'_{s_A} \quad \Sigma_2, \Delta'_{s_A} \vdash s_A \rightsquigarrow s'_A[active = \texttt{false}], \Delta''_{s_A}}{\Sigma_2, \Delta_{s_A} \vdash s_A \longrightarrow_{\downarrow} s'_A, \Delta''_{s_A}}$$

PTrigger of s_B for port p_B:

$$\frac{\Delta \vdash s_B.active \rightsquigarrow \texttt{true} \quad p_B \in s_B.ports : \quad \Sigma_2, \Delta_{s_B} \vdash p_B.cond \rightsquigarrow \texttt{true} \quad \dfrac{\dots}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow_{\downarrow} s'_B, \Delta''_{s_B}} \quad \Delta''_{s_B} \vdash p_B \rightsquigarrow p'_B[active = \texttt{true}], \Delta'''_{s_B}}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow s'_B, \Delta'''_{s_B}}$$

Deactivation of s_B:

$$\frac{\Delta_{s_B} \vdash s_B.active \rightsquigarrow \texttt{true} \quad s_B.children = \emptyset \quad \Sigma_2, \Delta_{s_B} \vdash s_B.exit \rightsquigarrow s_B.exit, \Delta'_{s_B} \quad \Sigma_2, \Delta'_{s_B} \vdash s_B \rightsquigarrow s'_B[active = \texttt{false}], \Delta''_{s_B}}{\Sigma_2, \Delta_{s_B} \vdash s_B \longrightarrow_{\downarrow} s'_B, \Delta''_{s_B}}$$

Micro 2-2: Application of PTrans1 on s_P with two active ports p_A and p_B. Even though evaluation order is not explicitly defined, this derivation evaluates p_A first. The contexts propagated up with $(\Delta''_{s_C}, \Delta'_{p_A}, \Delta''_{s_D}, \Delta'_{p_B}) \rightharpoonup \Delta''$. Note that the $s_P.active$ and action related premises are omitted.

$$\frac{\dfrac{\dfrac{\dots}{\Sigma_2, \Delta_{s_C}, p_A \vdash s_C \longrightarrow_{\uparrow} s'_C, \Delta''_{s_C}} \quad \Delta_{p_A} \vdash p_A \rightsquigarrow p'_A[\neg active], \Delta'_{p_A} \quad \dfrac{\dots}{\Sigma_2, \Delta_{s_D}, p_B \vdash s_D \longrightarrow_{\uparrow} s_D, \Delta''_{s_D}} \quad \Delta_{p_B} \vdash p_B \rightsquigarrow p'_B[\neg active], \Delta'_{p_B}}{(p_A, s_C) \in active\_ports(\Delta', s_P) : \ \Delta' \rightharpoonup (\Delta_{p_A}, \Delta_{s_C}) \qquad (p_B, s_D) \in active\_ports(\Delta', s_P) : \ \Delta' \rightharpoonup (\Delta_{p_B}, \Delta_{s_D})}}{\Sigma_2, \Delta' \vdash s_P \longrightarrow s_P, \Delta''}}{\Sigma_2 \vdash \Delta' \Longrightarrow \Delta''}$$

Activation of s_C (from port p_A):

$$\frac{\Delta_{s_C} \vdash s_C.active \rightsquigarrow \texttt{false} \quad \Delta_{s_C} \vdash s_C \rightsquigarrow s'_C[active = \texttt{true}], \Delta'_{s_C} \quad \Sigma_1, \Delta'_{s_C} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_C}}{\Sigma_2, \Delta_{s_C}, p_A \vdash s_C \longrightarrow_{\uparrow} s'_C, \Delta''_{s_C}}$$

Activation of s_D (from port p_B):

$$\frac{\Delta_{s_D} \vdash s_D.active \rightsquigarrow \texttt{false} \quad \Delta_{s_D} \vdash s_D \rightsquigarrow s'_D[active = \texttt{true}], \Delta'_{s_D} \quad \Sigma_1, \Delta'_{s_D} \vdash s.entry \rightsquigarrow s.entry, \Delta''_{s_D}}{\Sigma_2, \Delta_{s_D}, p_B \vdash s_D \longrightarrow_{\uparrow} s'_D, \Delta''_{s_D}}$$

Micro 2-3: PTrigger of s_P is applicable as the port condition of p_P is fulfilled

$$\frac{\Delta \vdash s_P.active \rightsquigarrow \texttt{true} \quad p_P \in s_P.ports: \quad \Sigma_2, \Delta_{s_P} \vdash p_P.cond \rightsquigarrow \texttt{true} \quad \dfrac{\overset{\ldots}{\Sigma_2, \Delta'' \vdash s_P \longrightarrow_{\downarrow} s'_P, \Delta'''} \quad \Delta''' \vdash p_P \rightsquigarrow p'_P[active = \texttt{true}], \Delta_2}{\Sigma_2, \Delta'' \vdash s_P \longrightarrow s'_P, \Delta_2}}{\Sigma_2 \vdash \Delta'' \Longrightarrow \Delta_2}$$

Deactivation of s_P, where only deactivation of s_C and s_D is included here, the context propagates up with $(\Delta''_{s_C}, \Delta''_{s_D}) \rightharpoonup \Delta'''$ and the active check of s_P is omitted.

$$\frac{\begin{array}{c}\overset{\ldots}{\Sigma_2, \Delta_{s_C} \vdash s_C \longrightarrow_{\downarrow} s'_C, \Delta''_{s_C}} \\ s_C \in s_P.children: \ \Delta'' \rightharpoonup \Delta_{s_C}\end{array} \quad \begin{array}{c}\overset{\ldots}{\Sigma_2, \Delta_{s_D} \vdash s_D \longrightarrow_{\downarrow} s'_D, \Delta''_{s_D}} \\ s_D \in s_P.children: \ \Delta'' \rightharpoonup \Delta_{s_D}\end{array} \quad \Sigma_2, \Delta''' \vdash s_P.exit \rightsquigarrow s_P.exit, \Delta'''' \quad \Sigma_2, \Delta'''' \vdash s_P \rightsquigarrow s'_P[\neg active], \Delta_2}{\Sigma_2, \Delta'' \vdash s_P \longrightarrow_{\downarrow} s'_P, \Delta_2}$$

Deactivation of s_C and s_D:

$$\frac{\Delta_{s_C} \vdash s_C.active \rightsquigarrow \texttt{true} \quad \Sigma_2, \Delta_{s_C} \vdash s_C.exit \rightsquigarrow s_C.exit, \Delta'_{s_C} \quad \Sigma_2, \Delta'_{s_C} \vdash s_C \rightsquigarrow s'_C[active = \texttt{false}], \Delta''_{s_C}}{\Sigma_2, \Delta_{s_C} \vdash s_C \longrightarrow_{\downarrow} s'_C, \Delta''_{s_C}}$$

$$\frac{\Delta_{s_D} \vdash s_D.active \rightsquigarrow \texttt{true} \quad \Sigma_2, \Delta_{s_D} \vdash s_D.exit \rightsquigarrow s_D.exit, \Delta'_{s_D} \quad \Sigma_2, \Delta'_{s_D} \vdash s_D \rightsquigarrow s'_D[active = \texttt{false}], \Delta''_{s_D}}{\Sigma_2, \Delta_{s_D} \vdash s_D \longrightarrow_{\downarrow} s'_D, \Delta''_{s_D}}$$