Formal Analysis of Variability-Intensive and Context-Sensitive Systems

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Philipp Chrszon
geboren am 17. Juli 1989 in Zwickau

Gutachter:

Prof. Dr. rer. nat. Christel Baier
Technische Universität Dresden

Prof.dr. Frank S. de Boer
Universität Leiden, Niederlande

Tag der Verteidigung: 28. September 2020

Abstract

With the widespread use of information systems in modern society comes a growing demand for customizable and adaptable software. As a result, systems are increasingly developed as families of products adapted to specific contexts and requirements. Features are an established concept to capture the commonalities and variability between system variants. Most prominently, the concept is applied in the design, modeling, analysis, and implementation of software product lines where products are built upon a common base and are distinguished by their features. While adaptations encapsulated within features are mainly static and remain part of the system after deployment, dynamic adaptations become increasingly important. Especially interconnected mobile devices and embedded systems are required to be context-sensitive and (self-)adaptive. A promising concept for the design and implementation of such systems are roles as they capture context-dependent and collaboration-specific behavior.

A major challenge in the development of feature-oriented and role-based systems are interactions, i.e., emergent behavior that arises from the combination of multiple features or roles. As the number of possible combinations is usually exponential in the number of features and roles, the detection of such interactions is difficult. Since unintended interactions may compromise the functional correctness of a system and may lead to reduced efficiency or reliability, it is desirable to detect them as early as possible in the development process.

The goal of this thesis is to adopt the concepts of features and roles in the formal modeling and analysis of systems and system families. In particular, the focus is on the quantitative analysis of operational models by means of probabilistic model checking for supporting the development process and for ensuring correctness.

The tool ProFeat, which enables a quantitative analysis of stochastic system families defined in terms of features, has been extended with additional language constructs, support for a one-by-one analysis of system variants, and a symbolic representation of analysis results. The implementation is evaluated by means of several case studies which compare different analysis approaches and show how ProFeat facilitates a family-based quantitative analysis of systems.

For the compositional modeling of role-based systems, role-based automata (RBA) are introduced. The thesis presents a modeling language that is based on the input language of the probabilistic model checker PRISM to compactly describe RBA. Accompanying tool support translates RBA models into the PRISM language to enable the formal analysis of functional and non-functional properties, including system dynamics, contextual changes, and interactions. Furthermore, an approach for a declarative and compositional definition of role coordinators based on the exogenous coordination language Reo is proposed. The adequacy of the RBA approach for detecting interactions within context-sensitive and adaptive systems is shown by several case studies.

Acknowledgements

First of all I want to thank my advisor *Christel Baier* for giving me the opportunity to join her group as a PhD student and for her continued support over the years. Even in the most stressful times she always took the time for meetings and discussions which I highly appreciate. Her ability to provide a new perspective and to broaden my horizon when I was stuck on a problem really helped to shape and focus the ideas that finally made it into this thesis.

A special thanks goes to *Clemens Dubslaff* and *Sascha Klüppelholz* who mentored, supported, and encouraged me especially at the beginning of my journey as a PhD student. They were always open for any discussion (and I mean *any* discussion) and taught me a lot about the craft of scientific writing. Furthermore, I would like to thank my former and present colleagues *Marcus Daum*, *Florian Funke*, *Daniel Gburek*, *Linda Herrmann*, *Lisa Hutschenreiter*, *Joachim Klein*, *Max Korn*, *David Müller*, *Jakob Piribauer*, *Sandy Seifarth*, *Karina Wauer*, *Sascha Wunderlich*, and in particular my officemates *Steffen Märcker* and *Simon Jantsch* for the very special and friendly atmosphere in our group.

As a member of the RoSI research training group, I also want to thank my RoSI-colleagues *Stephan Böhme*, *İsmail İlkan Ceylan*, *Kai Herrmann*, *Tobias Jäkel*, and *Johannes Bamme* who shared the big RoSI office with me during my first three years and made me feel welcome from the very first day.

During my visit at CWI in Amsterdam I had the pleasure of meeting *Farhad Arbab*, with whom I had many interesting discussions, and also *Kasper Dokter* who introduced me to some newer developments around Reo, specifically the ReoCompiler, which inspired one of the chapters of this thesis. I also want to thank *Benjamin Lion* for sharing his office with me during my stay.

Furthermore, I want to thank Simon, Steffen, Clemens, and *Johannes Fenzl* for proofreading parts of this thesis.

Finally, I want to thank my family and friends. I owe special gratitude to my loving parents for their support and encouragement, especially during the final phase of writing.

Contents

1 Introduction 1
- 1.1 Engineering approaches for variant-rich adaptive systems 2
- 1.2 Validation and verification methods 4
- 1.3 Analysis of feature-oriented and role-based systems 6
- 1.4 Contribution 7
- 1.5 Outline 9

2 Preliminaries 11

I Feature-oriented systems 13

3 Feature-oriented engineering for family-based analysis 15
- 3.1 Feature-oriented development 16
- 3.2 Describing system families: The ProFeat language 20
  - 3.2.1 Feature-oriented language constructs 21
  - 3.2.2 Parametrization 25
  - 3.2.3 Metaprogramming language extensions 26
  - 3.2.4 Property specifications 27
  - 3.2.5 Semantics 28
- 3.3 Implementation 29
  - 3.3.1 Translation of ProFeat models 32
  - 3.3.2 Post-processing of analysis results 36

4 Case studies and application areas 41
- 4.1 Comparing family-based and product-based analysis 42
  - 4.1.1 Analysis of feature-oriented systems 42
  - 4.1.2 Analysis of parametrized systems 45
- 4.2 Software product lines 46
  - 4.2.1 Body sensor network 47
  - 4.2.2 Elevator product line 50
- 4.3 Self-adaptive systems 51
  - 4.3.1 Adaptive network system model 52
  - 4.3.2 Adaptation protocol for distributed systems 56

II Role-based systems 61

5 Formal modeling and analysis of role-based systems 63
5.1 The role concept 63
5.1.1 Towards a common notion of roles 64
5.1.2 The Compartment Role Object Model 67
5.1.3 Roles in programming languages 69
5.2 Compositional modeling of role-based behavior 70
5.2.1 Role-based automata and their composition 72
5.2.2 Algebraic properties of compositions 77
5.2.3 Coordination and semantics of RBA 80

6 Implementation of a role-oriented modeling language 85
6.1 Role-oriented modeling language 85
6.1.1 Declaration of the system structure 86
6.1.2 Definition of operational behavior 94
6.2 Translation of role-based models 106
6.2.1 Transformation to multi-action MDPs 107
6.2.2 Multi-action extension of PRISM 115
6.2.3 Translation of components 118
6.2.4 Translation of role-playing coordinators 120
6.2.5 Encoding role-playing into states 122

7 Exogenous coordination of roles 127
7.1 The exogenous coordination language Reo 128
7.2 Constraint automata 130
7.3 Embedding of role-based automata in constraint automata 132
7.4 Implementation 135
7.4.1 Exogenous coordination of PRISM modules 135
7.4.2 Reo for exogenous coordination within PRISM 136

8 Evaluation of the role-oriented approach 139
8.1 Experimental studies 139
8.1.1 Peer-to-peer file transfer 140
8.1.2 Self-adaptive production cell 148
8.1.3 File transfer with exogenous coordination 151
8.2 Classification 154
8.3 Related work 158
8.3.1 Role-based approaches 158
8.3.2 Aspect-oriented approaches 161
8.3.3 Feature-oriented approaches 162

9 Conclusion 165

List of Figures 169

List of Tables 171

List of Listings 173

Acronyms 175

Bibliography 177

Publications

Philipp Chrszon, Clemens Dubslaff, Christel Baier, Joachim Klein, and Sascha Klüppelholz. "Modeling Role-Based Systems with Exogenous Coordination". In: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. 2016, pp. 122–139. doi: 10.1007/978-3-319-30734-3_10.

Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. "Family-Based Modeling and Analysis for Probabilistic Systems - Featuring ProFeat". In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 287–304. doi: 10.1007/978-3-662-49665-7_17.

Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, and David Müller. "Advances in Symbolic Probabilistic Model Checking with PRISM". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 349–366. doi: 10.1007/978-3-662-49674-9_20.

Martin Weißbach, Philipp Chrszon, Thomas Springer, and Alexander Schill. "Decentrally Coordinated Execution of Adaptations in Distributed Self-Adaptive Software Systems". In: *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18-22*. 2017, pp. 111–120. doi: 10.1109/SASO.2017.20.

Christel Baier, Philipp Chrszon, Clemens Dubslaff, Joachim Klein, and Sascha Klüppelholz. "Energy-Utility Analysis of Probabilistic Systems with Exogenous Coordination". In: *It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab*. 2018, pp. 38–56. doi: 10.1007/978-3-319-90089-6_3.

Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. "ProFeat: feature-oriented engineering for family-based probabilistic model checking". In: *Formal Aspects of Computing* 30.1 (2018), pp. 45–75. doi: 10.1007/s00165-017-0432-4.

Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, and David Müller. "Advances in Probabilistic Model Checking with PRISM: Variable Reordering, Quantiles and Weak Deterministic Büchi

Automata". In: *International Journal on Software Tools for Technology Transfer* 20.2 (2018), pp. 179–194. DOI: 10.1007/s10009-017-0456-3.

Philipp Chrszon, Christel Baier, Clemens Dubslaff, and Sascha Klüppelholz. "From features to roles". In: *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020*. ACM, 2020, 19:1–19:11. DOI: 10.1145/3382025.3414962.

1 Introduction

Ever since the beginnings of the digital revolution, information systems have entered more and more areas of everyday life. Both society and industry are increasingly dependent on software and this trend steadily continues. With the widespread use of software comes a growing demand for specialized and customized software products as every company and every user has different needs, requirements, and a different budget. A simple example are software packages that come in a basic and a professional version which allow users to choose additional functionalities in exchange for a higher price. Moreover, software, services, and products deployed globally must often be adapted in order to conform with local regulations or cultural preferences. For instance, shopping websites must handle different currencies, address formats, shipping options, tax regulations, etc., all depending on the country of operation. As a result, many systems are provided as product families comprising a multitude of variants adapted to different requirements and contexts. While adaptations are mainly static, i.e., they remain a part of the software system once they have been implemented, dynamic adaptations become increasingly important. Especially interconnected systems that interact with the physical world, e.g., mobile devices and embedded systems, are expected or even required to be context-sensitive and (self-)adaptive. Adaptations may be triggered by changes in the environment or changing availability of resources, such as network connectivity, remaining battery power, as well as locally accessible services, devices, and sensors. Moreover, adaptivity is key in systems possessing self-healing and self-optimization capabilities [Bab+05] where the system adapts to recover from hardware- or software failures and where adaptations aim to improve, e.g., throughput, latency, or energy efficiency. Furthermore, adaptation mechanisms may also be utilized to integrate new functionality in long-running software systems. Adaptations, both static and dynamic, context-dependency, and distributed computation across multiple devices are major factors that drastically increase the complexity of modern software systems. Motivated by this rising complexity, several software-engineering approaches tailored to the development of variability-intensive and adaptive systems have been introduced. The goal of this thesis is to adopt these techniques also in the formal modeling and analysis of systems. In particular, the focus is on a quantitative analysis by means of probabilistic model checking for supporting the development process and for ensuring correctness. Important concepts for capturing variability and adaptivity are features and roles which will be discussed in the following.

1.1 Engineering approaches for variant-rich adaptive systems

Features are an established concept for describing the commonalities and variability among variants in the design, modeling, analysis, and implementation of system families. The concept is commonly applied in the feature-oriented development of *software product lines* (SPLs) [AK09; CE00; CN02; Gri00]. The individual products, i.e., variants, of a product line are built on a common base and are distinguished by their features. Here, a feature is generally considered to be a "user-visible aspect, quality, or characteristic" of the system [Kan+90] or an "optional or incremental unit of functionality" [Zav03]. The feature-oriented development approach promotes a variability-aware structuring of the system as well as a systematic reuse of assets and components. Feature-oriented programming is a paradigm that facilitates the modularization and encapsulation of feature implementations in feature modules. This enables an automatic generation of products from a set of feature modules and an implementation of the base functionality. The modularization of feature implementations is especially important for features that correspond to cross-cutting concerns [Kic+97] where a feature's implementation is scattered across multiple different components. Common examples of cross-cutting concerns are monitoring, logging, transaction management, synchronization, and caching. The inadequacy of traditional object-oriented design for modularizing cross-cutting concerns has led to the introduction of aspect-oriented programming [Kic+97] where aspects encapsulate modifications to existing components. The concepts and techniques provided by aspect-oriented programming are also applicable for feature-oriented development [Ape07]. Tool support for feature-oriented programming is provided by, e.g., GenVoca [BO92], the AHEAD tool suite [Bat04], and FEATUREHOUSE [AKL09]. The feature concept has found broad application in both academia and industry [SPL; Wei08] for the design, analysis, and implementation of system families.

For expressing the adaptivity of a system, the feature concept is applicable as well. The set of features is usually assumed to be static and does not change after deployment. In a *dynamic software product line* (DSPL) [GH03], the configuration, i.e., the feature combination, can be changed at runtime. The DSPL approach is employed to enable software upgrades after the initial deployment, e.g., from a basic to a professional version. Dynamic features are also suitable for modeling and implementing adaptive systems, as they can be activated and deactivated at runtime to react to different contexts or changes in the environment [Ach+09; Kru+15]. A language-level approach for context-dependent adaptivity is context-oriented programming [HCN08] which provides a first-class representation of context. Here, context-specific adaptations are captured in layers which can be dynamically activated and deactivated on context changes. Activating a layer automatically applies all contained adaptations to their respective components. Another concept that facilitates behavioral changes and adaptivity are *roles*. The role concept

has first been introduced by Bachman and Daya [BD77] to capture evolving objects and context-related information in data models. Inspired by the theatrical context, the role notion is intuitively understood and has subsequently been applied in various fields [ZZ08], including conceptual modeling [Küh+14; Ste00], programming languages, multi-agent systems [Cab+10], process modeling, architectural modeling [AG97], and computer-supported collaborative work. Roles are also prominently applied in role-based access control (RBAC) [FK92]. However, within RBAC the focus is on the description of security policies, not on dynamic adaptations. Surveys of role-based approaches in conceptual modeling and programming languages have been conducted by Steimann [Ste00] and Kühn et al. [Küh+14; Küh17], showing that there is no commonly accepted notion of roles. Roles have been considered to be defined by relationships between entities, as a collection of extrinsic properties, or as a collection of context-dependent attributes and behavior. A widely accepted trait of roles in conceptual modeling and programming languages is that they are able to modify the behavior of their player [Küh+14; Ste00]. As such, roles can be considered to be an abstraction of context-dependent and collaboration-specific behavior. Therefore, several role-based approaches consider context as a first-class concept [Gen07; Her02; HK14; Küh+15]. As the term context is both vague and overloaded, Kühn et al. [Küh+14] proposed the term *compartment* to refer to the objectified context in which roles are played. Several role-based programming languages (mostly extensions of Java) were proposed in the literature, e.g., JavaStage [BA12], powerJava [BBT06], Chameleon [GØ03], ObjectTeams/Java [Her02], Rava [He+06], and EpsilonJ [TUI05]. On a technical level, there are several approaches that can be employed to implement the adaptation mechanisms for dynamic features and roles, e.g., metaprogramming with reflection [Mae87], dynamic delta-oriented programming [DS11], and dynamic aspect weaving [Boc+04; Din+10; SCT03]. In this thesis, the feature concept as well as the role concept are used to capture variability and context-dependent adaptations within formal operational models.

Even though features and roles are conceptually quite different, i.e., features describe variability and roles capture context-dependent or collaboration-specific behavior, they share some notable similarities from the perspective of behavioral adaptation. The different roles a player might acquire can be considered to be features of this player that correspond to its context-specific behavior. The role concept admits a distinction between role-binding and role-playing [MKK12]. Role-binding enables a player to play the bound role, and role-playing corresponds to an active enactment of the role behavior. For instance, a person might become a teacher (binding) and then actually teaches a class (playing the role of a teacher). Role-binding corresponds to the composition of a feature with the system and role-playing corresponds to the activation of a feature and subsequent execution of the feature behavior.

A major challenge in the development of feature-oriented systems are feature interactions [Ape+11; Ape+13a; PR01] which refer to the emergent behavior that arises from the

combination of two or more features. While feature interactions may be intended, as they allow for a modularization of the system's functionalities, they may also be unintended, leading to unforeseen side effects. Since the number of possible feature combinations is usually exponential in the number of features, the feature-interaction problem, i.e., the detection of (unintended) interactions, is challenging. Clearly, the same is true for role-based systems where interactions between roles may occur and where the number of possible role combinations might also be exponential. The difficulty of detecting interactions is further amplified when considering dynamic features and role-playing since the temporal order of feature activation and role-playing may have an influence on the arising interactions.

Unintended interactions have a substantial impact on the correctness of a system. It is thus desirable to detect interactions as early as possible in the development process. For that, various validation, verification, and analysis methodologies may be applied. An overview of these methods is provided in the next section.

1.2 Validation and verification methods

While the complexity of information systems steadily increases, the correctness of these systems becomes more and more important. Software errors, especially in critical infrastructure, are not only costly, but may actually endanger human lives. Besides functional correctness, the non-functional, i.e., quantitative, aspects of a system are often crucial as well. Especially embedded systems usually must operate under strict real-time constraints. Another example is software on mobile, battery-powered devices which should not only work correctly, but should also be energy-efficient. Within aerospace applications, resilience and the mean time between failures are important properties of systems. In addition to checking a system's conformance to certain non-functional properties, a quantitative analysis may also be applied to optimize the configuration of a system, e.g., by parameter tuning or selection of an optimal variant w.r.t. some optimization criterion. Furthermore, a cost-utility analysis can reveal the trade-offs in the configuration of the system, e.g., between energy-consumption and throughput. Given the increasing importance of correctness, various methodologies and techniques for establishing functional and non-functional properties of systems have been established.

Generally, there are several approaches and techniques to validate and verify the correctness of software systems (not limited to feature-oriented and role-based systems). *Testing* [Bei90; MSB11] and *simulation* of systems and system models, respectively, usually involve running the system and checking for unexpected behavior or outputs. As full coverage of all possible behaviors is rarely achieved, these methods usually cannot show the absence of errors. Testing and simulation are complemented by the formal verification methods *theorem proving* [BC04; NK14] and *model checking* [BK08; CGP01]. These methods are complete since properties that have been proven correct hold for

all possible executions of the system or model. Theorem proving is a semi-automatic approach where nontrivial proof steps have to be supplied by the user, usually in an interactive session with a proof assistant [BC04; NK14]. In contrast, model checking is a fully automatic verification technique that systematically explores the state space of a system without requiring any user input. This thesis focuses on model checking for the verification and analysis of systems as this methodology is particularly well-suited for concurrent systems.

Model checking has been introduced independently by both Clarke and Emerson [CE81] as well as Queille and Sifakis [QS82] in the early eighties. Generally speaking, the analyzed system is formalized as an automata-based model where transitions between the system's states define the operational behavior. Requirements are given in terms of a temporal logic formula, e.g., using *linear temporal logic* (LTL) [Pnu77] or *computation-tree logic* (CTL) [CE81]. Then, a model checker, i.e., a tool that implements the appropriate model-checking algorithm, automatically checks whether the model satisfies the specification. In case the answer is negative, usually an explanation in the form of a counterexample, e.g., a path to an invalid state, is returned which may provide insights on how to amend the model or the specification. A comprehensive introduction of model checking is provided in the books [CGP01] and [BK08]. Different model checking approaches are mainly distinguished by the underlying model type and the specification formalism for which they are applicable. Most commonly, transition systems [Kel76] or Kripke structures serve as model representation. Real-time systems can be described using timed automata [AD90; LPY95]. For probabilistic model checking, Markovian models are applied to reason about the likelihood of certain events for answering questions such as: "Is the probability that a message is transmitted successfully within 5 steps greater than 0.95?". A Markov chain can be seen as a transition system where transitions are purely probabilistic. The combination of probabilistic choice and nondeterminism is realized by *Markov decision processes* (MDPs) [Var85] where the nondeterministic selection of an action is followed by a probabilistic choice of the successor state. With that, MDPs are an appropriate model to capture both concurrent processes (by nondeterministic selection of the next scheduled process) and stochastic effects within the modeled system itself as well as in the system's environment.

Since model-checking algorithms operate on the representations of whole systems, the corresponding models may become prohibitively large. For instance, the model size is generally exponential in the number of program variables and the number of parallel processes. This issue is known as the *state-space explosion* problem. Several techniques for tackling this problem have been developed, e.g., assume-guarantee reasoning [HQR98; Pnu84], partial order reduction [God96; Pel93; Val92], symmetry reduction [Cla+96; DM06], and SAT-based model checking [Bie+99; Wil+00]. Another approach is the use of symbolic model representations [Bur+92; McM93]. Instead of explicitly representing each individual state of the system in memory, *binary decision diagrams* (BDDs) [Bry86] are

employed to compactly represent sets of states. Using this technique, even the analysis of very large system models becomes tractable [Bur+92].

Several model-checking tools have been developed. The explicit-state model checker SPIN provides the input language Promela for modeling systems comprising parallel processes communicating over shared variables and bounded channels. Promela models have a transition-system semantics and can be checked against LTL specifications or monitor automata. SMV [McM93] and its successor NuSMV [Cim+00] apply symbolic model checking for verifying systems under CTL specifications. While SPIN and NuSMV provide input languages tailored towards the definition of transition systems, there are also model checking tools working directly on the source code of general-purpose programming languages. An example is Java PathFinder [HP00] which translates Java programs into Promela for a subsequent analysis using SPIN. Later versions implement a Java virtual machine with model checking capabilities that is able to directly check Java programs without a translation step. Checking of safety properties on C programs is provided by the model checker BLAST [Bey+07]. UPPAAL [Ben+95] supports the verification of real-time systems expressed as a network of timed automata. Probabilistic model checking can be carried out using, e.g., MRMC [Kat+11], PRISM [KNP11], the Modest tool set [HH14], iscasMc [Hah+14], or Storm [Hen+20]. Since PRISM is a state-of-the-art model checker and its input language is also supported by other tools, e.g., Storm and iscasMc, it provides the basis for the tooling presented in this thesis.

Testing, theorem proving, and model checking are immediately applicable for analyzing feature-oriented systems. However, this requires generating all variants of the system family which are then subsequently analyzed one by one. Because of the exponential number of variants in the number of features, this is infeasible for families with many features. Furthermore, features are only implicitly encoded in the system variants which makes the identification of feature interactions challenging. Making features explicit in formal modeling enables a feature-aware analysis that can pinpoint problematic feature combinations directly. Moreover, providing features as a first-class modeling concept allows for a succinct and modular description of model families. Similarly, adopting the role concept in a formal modeling approach facilitates a modularization and a separation of concerns also on the model level by encapsulating context-specific behavior within roles. A role-aware analysis does not only make interactions between roles explicit, but also the context in which they occur. The following section gives an overview of approaches towards feature-oriented as well as role-based modeling and analysis.

1.3 Analysis of feature-oriented and role-based systems

Software systems with high variability, i.e., system families with a high number of variants, are already applied in within critical infrastructure and services, such as medical

applications, telecommunication, avionics, engine control systems, and factory management [SPL; Wei08]. Thus, the correctness of these systems is imperative. A wide range of feature-aware modeling and analysis approaches have been introduced. These approaches typically try to exploit the structure of feature-oriented systems as well as the commonalities between variants to considerably speed up the analysis. For variability-aware model checking, *featured transition systems* (FTS) [Cla+10; Cla+13] have been proposed. Within an FTS, transitions are guarded by feature combinations which allows the definition of a whole system family using a single model. Based on FTS, the model checker SNIP [Cla+12; Cla+13] and its re-engineered successor ProVeLines [Cor+13b] have been developed. Their input language is based on Promela, extended with feature-related constructs. Both tools allow the verification of whole system families without generating the individual variants, and return the set of family instances violating a given property. Symbolic model checking of feature-oriented systems has been implemented as a modest extension of NuSMV [Cla+14]. None of the previously mentioned variability-aware tools deals with dynamic features or stochastic behavior. The compositional modeling framework for feature-oriented systems proposed by Dubslaff et al. [DBK15] uses an MDP-like formalism to model the operational behavior of features. The framework also allows for modeling dynamic feature-oriented systems using an additional automata-based component called feature controller that defines the rules for the activation and deactivation of features. The semantics of a set of features under a feature controller is given in terms of an MDP. A detailed discussion of further feature-related analysis approaches and corresponding tools is provided in Section 3.1. A survey of formal analysis approaches for feature-oriented systems has been conducted by Thüm et al. [Thü+14a]. As the approach of [DBK15] supports dynamic features, stochastic effects, cost annotations, and is compositional, it will serve as a basis for the feature-oriented analysis approach presented in this thesis.

Formal modeling and analysis approaches tailored specifically towards the concept of roles for capturing context-dependent behavior have been given little attention in the literature. Notably, the Helena approach [HK14] utilizes transition systems for modeling the operational behavior of collaborating, role-playing components. An automated translation [Kla15] of Helena models into Promela enables a verification using SPIN. However, this approach does not consider behavioral adaptations of players by their respective roles and it also does not cover stochastic behavior. An in-depth discussion of other related approaches for capturing adaptations in operational models is provided in Section 8.3.

1.4 Contribution

This thesis introduces formal approaches for modeling and analyzing variant-rich adaptive systems by means of probabilistic model checking where variability is expressed using features and context-dependent behavior is captured by roles, respectively. Models

can be defined compositionally and may be annotated with probabilities and costs for a quantitative analysis. An analysis of such systems is enabled by tool support that leverages the probabilistic model checker PRISM. In particular, the contributions of this thesis are the following:

1. The tool ProFeat implements the formal framework introduced by Dubslaff et al. [DBK15] and provides an automated translation of a feature-oriented modeling language into the input language of PRISM. The tool was initially developed as part of my master thesis [Chr14] and has been extended as part of this thesis. Constructs for defining system families in terms of parameters and feature attributes have been added to the ProFeat language. The implementation has been extended with support for a one-by-one analysis of system families. Furthermore, a feature-aware post-processing of the analysis results was added, including support for the transformation of results into compact symbolic representations.

2. To show the practical applicability of ProFeat, several case studies have been conducted. The first set of case studies compares the all-in-one and one-by-one analysis approaches for different kinds of models and scenarios. Additional case studies show the applicability of ProFeat for analyzing both static and dynamic SPLs. Moreover, the potential of using feature-oriented modeling for defining adaptive systems is illustrated.

3. This thesis proposes *role-based automata* (RBA) which provide roles as a first-class construct for modeling the operational behavior of systems. Accompanying composition operators for parallel composition and role-binding enable a compositional representation of role-specific behavior and facilitate the coordination of role-playing. The framework provides an MDP semantics of role-playing components under a role-playing coordinator, enabling a quantitative analysis of role-based systems.

4. A component-based and role-oriented modeling language with RBA semantics is presented. It extends the PRISM language with a role-aware type system as well as constructs for defining role-specific behavior and the coordination of roles.

5. An automated translation of the role-oriented modeling language into the input language of PRISM is provided by a dedicated tool. The translation approach utilizes MDPs with multi-actions [Bai+18] to avoid an exponential blow-up of the translated model's size.

6. The potential of detecting interactions both within and also between dynamic self-adaptive systems is shown by means of two experimental studies.

7. The coordination of role-playing using an exogenous coordination language is proposed. This approach enables a compositional and hierarchical specification of role coordination.

1.5 Outline

Chapter 2 briefly covers the relevant definitions and notations that are used throughout the thesis. As feature-oriented systems and role-based systems are addressed by separate approaches, the thesis is accordingly divided into two parts. The first part starts with an overview of the feature-oriented development approach and a discussion of existing model-checking approaches tailored towards system families defined in terms of features. Section 3.2 and Section 3.3 introduce the ProFeat language and implementation, respectively. The applicability of the feature-oriented modeling and analysis approach is shown in Chapter 4 which completes the first part of the thesis. Chapter 5 starts the second part with an introduction of the role concept, focusing on conceptual modeling and role-based programming languages. In Section 5.2, the first main contribution of this part, the RBA formalism, is presented. Afterwards, the role-oriented modeling language as well as the translational approach of the implementation are described in Chapter 6, followed by Chapter 7 which proposes an approach for the exogenous coordination of role-playing. Chapter 8 concludes the second part of the thesis by evaluating the role-oriented analysis approach by three experimental studies (Section 8.1), by classifying the approach (Section 8.2), and by relating it to existing work (Section 8.3). Finally, the thesis is concluded in Chapter 9.

2 Preliminaries

In the following, relevant notations and definitions are introduced.

The set of natural numbers including zero is denoted by $\mathbb{N}$. The *power set* of a set X is denoted by $\mathcal{P}(X)$.

Distributions. If X is a countable, nonempty set of outcomes, then a (probability) *distribution* on X is a function $\lambda : X \to [0,1]$ such that $\sum_{x \in X} \lambda(x) = 1$. $\mathit{Distr}(X)$ denotes the set of all distributions over X. Given some $x \in X$, the *Dirac distribution* $\mathit{Dirac}(x)$ over X is defined by $\mathit{Dirac}(x)(x) = 1$ and $\mathit{Dirac}(x)(y) = 0$ for all $y \in X \setminus \{x\}$. The *product* of two distributions $\lambda_1 \in \mathit{Distr}(X_1)$ and $\lambda_2 \in \mathit{Distr}(X_2)$ is the distribution $\lambda_1 * \lambda_2 \in \mathit{Distr}(X_1 \times X_2)$ where for all $x_1 \in X_1$ and $x_2 \in X_2$ we have $(\lambda_1 * \lambda_2)(x_1, x_2) = \lambda(x_1) \cdot \lambda(x_2)$.

Markov decision processes. An MDP is a tuple $\mathcal{M} = (S, \mathit{Act}, \longrightarrow, S^{init})$ where S is a finite set of states, Act is a finite set of actions, $\longrightarrow \subseteq S \times \mathit{Act} \times \mathit{Distr}(S)$ is the transition relation, and $S^{init} \subseteq S$ is the set of initial states. We use the notation $s \xrightarrow{\alpha} \lambda$ for $(s, \alpha, \lambda) \in \longrightarrow$. A *reward* function $\mathit{rew} : S \times \mathit{Act} \to \mathbb{N}$ assigns rewards (or costs) to transitions.

The intuitive behavior of an MDP is as follows. First, an initial state from the set S^{init} is chosen nondeterministically. Within some state s, there is a nondeterministic choice of an action α with $s \xrightarrow{\alpha} \lambda$, followed by a probabilistic choice of a successor state $s' \in S$ with $\lambda(s') > 0$.

The *Paths* of an MDP $\mathcal{M}$ are finite or infinite sequences of alternating states and actions $s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 \ldots$ where $s_i \xrightarrow{\alpha_i} \lambda$ and $\lambda(s_{i+1}) > 0$ for all $i \geq 0$. The set of finite paths is denoted by *FPaths*. Given a finite path $\pi = s_0\alpha_0 s_1\alpha_1 \ldots \alpha_{k-1} s_k$, the accumulated reward along π is defined as $\mathit{rew}(\pi) = \mathit{rew}(s_0, \alpha_0) + \mathit{rew}(s_1, \alpha_1) + \ldots + \mathit{rew}(s_{k-1}, \alpha_{k-1})$.

Reasoning about probabilities in MDPs requires the resolution of the nondeterministic choice between transitions. This is formalized via *schedulers* which take a finite path and select the next action to be taken. In this thesis, we only consider deterministic schedulers. Formally, such a scheduler is a function $\mathfrak{S} : \mathit{FPaths} \to \mathit{Act} \times \mathit{Distr}(S)$. If π is a finite path and $\mathfrak{S}(\pi) = (\alpha, \lambda)$ then there is a transition $s \xrightarrow{\alpha} \lambda$ where s is the last state in π. After selecting an initial state $s \in S^{init}$, the behavior of an MDP under a scheduler $\mathfrak{S}$ is purely probabilistic and corresponds to an infinite-state, tree-like Markov chain $\mathcal{M}_s^{\mathfrak{S}}$. Using standard concepts of measure theory, a probability measure $\Pr_s^{\mathfrak{S}}$ for measurable sets of

infinite paths in the Markov chain $\mathcal{M}_s^{\mathfrak{S}}$ can be defined. Further details are provided by standard text books, e.g., [Kul16; Put94].

Structured operational semantics. Throughout this thesis, transition relations are often defined using the so-called SOS-notation [Plo04]. An SOS-rule (also called *inference rule*) has the following form:

$$\frac{\textit{premise}}{\textit{conclusion}}$$

If the *premise* above the line holds, the *conclusion* below holds as well.

Part I

Feature-oriented systems

3 Feature-oriented engineering for family-based analysis

Feature-oriented development is an established approach for the design and implementation of system families. Features provide an abstraction of the variability among a set of similar systems built on a common base by defining the changes that are applied to the base system. Since features promote a systematic reuse of components, they facilitate a family-based analysis that exploits the commonalities among the individual members of the system family. The most prominent application of the feature concept is the development of SPLs [AK09; CE00; CN02; Gri00].

This chapter presents the tool ProFeat for the quantitative analysis of (stochastic) system families by means of probabilistic model checking. The ProFeat modeling language extends the input language of the probabilistic model checker PRISM [KNP11] with feature-related constructs, including support for dynamic features, multi-features, and feature attributes. For analyzing a system family defined in terms of features, ProFeat translates the corresponding model into the PRISM language and utilizes PRISM for carrying out the analysis. The analysis results can be converted into a concise symbolic representation which aids to identify problematic feature combinations or may guide the feature selection, and thus supports the feature-oriented development process.

The work presented in this chapter has been published in [Chr+16b] and [Chr+18]. The initial version of the ProFeat language and the corresponding tool have been conceptualized and implemented in my master thesis [Chr14]. Since then, the language has been extended with the following constructs. First, support for feature attributes and with that support for numerical constraints over both features and attributes was added. Second, the blocking behavior of inactive features has been made configurable, thereby lifting some restrictions imposed by the first version of the language. Third, a construct for modeling system families defined by one or more parameters has been added. The implementation has also been extended in several directions. The original implementation of ProFeat only supported an all-in-one analysis where all members of a system family are analyzed at once. The extended version presented in this thesis also supports a one-by-one analysis where system family instances are analyzed individually. Finally, the facilities for result post-processing, including the conversion to symbolic representations, have been added.

Outline. The first section of this chapter gives an overview of the feature-oriented development process and product-line development. Furthermore, the related work regarding the verification and analysis of feature-oriented systems by means of model checking is discussed. Section 3.2 describes the ProFeat language with a focus on the feature-related modeling constructs. Finally, an overview of the implementation and notable implementation details are provided in Section 3.3.

3.1 Feature-oriented development

The feature concept is mainly applied within the development of SPLs [AK09; CE00; CN02; Gri00]. A software product line comprises a family of similar systems (called *products*) that are built upon a common base. The products differ in the set of features they provide. Thus, features are an abstraction of the *variability* in a system family. Generally, a *feature* is an "optional or incremental unit of functionality" [Zav03]. A feature may also have an impact on the non-functional properties of a system, e.g., performance, energy efficiency, or reliability. In software product line engineering, features promote a *systematic reuse* of assets [CE00], such as implementation artifacts or documentation, which enables an efficient development of system families. New products do not have to be created from scratch, but can be assembled from already implemented parts. This allows users to customize a product to fit their specific needs. Furthermore, it enables the reaction to changing demands of stakeholders or the system's environment.

Usually, not all combinations of features are allowed or possible. For instance, if there is a standard and a professional version of some feature, then both cannot be combined in the same product. A *feature model*, also called *variability model*, defines which feature combinations are *valid*, i.e., allowed, and thus defines the set of products. There are several possible representations of feature models. The simplest one is a list of all valid feature combinations. However, since the number of possible feature combinations is exponential in the number of features, this representation is only feasible for product lines with only a few features or products.

A compact graphical representation of the valid feature combinations is provided by *feature diagrams* [Kan+90]. An example of a small car product line is depicted in Figure 3.1. In a feature diagram, features are organized in a hierarchical tree-like structure where each sub-feature depends on its parent feature. For instance, the Body feature can only be part of the feature combination if its parent feature Car is included as well. The root feature, also called *concept*, is contained in every feature combination. If a feature is marked as mandatory, it must be selected whenever its parent feature is selected[1]. In the example, every car must have an Engine, but not every car needs to be equipped with a sunroof since the Sunroof feature is marked as optional. The model elements for "or" (filled arc)

[1]In this thesis, we assume "mandatory" to be the default case and drop the markers in subsequent feature diagrams.

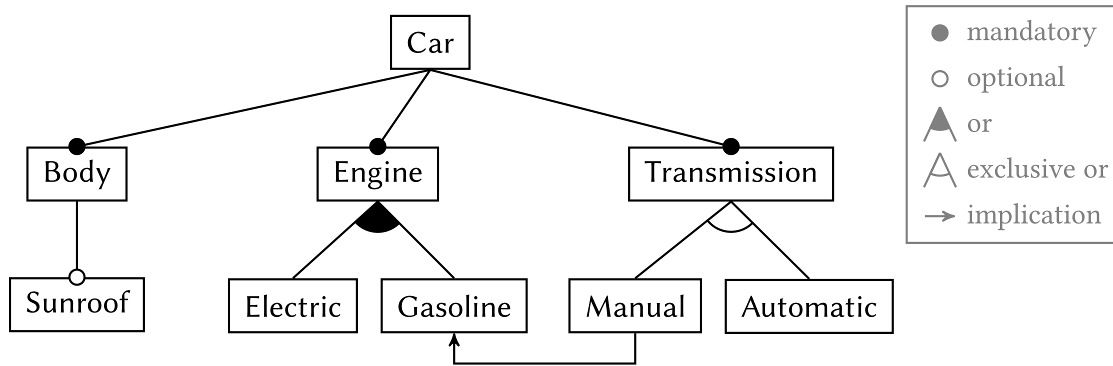


Figure 3.1: Feature diagram for a simple car product line

and "exclusive or" (non-filled arc) express that at least on sub-feature must be included and that exactly one sub-feature must be selected, respectively. For instance, a car must have at least an Electric engine or a Gasoline engine, but hybrid vehicles may have both. Furthermore, cross-tree constraints can be added to further restrict the set of feature combinations. In the example, cars with Manual transmission must have a Gasoline engine. The basic feature diagram notation originally introduced within the feature-oriented domain analysis (FODA) [Kan+90] has been extended to improve the expressiveness and to represent large product lines more concisely. Hierarchical constraints have been generalized to UML-like multiplicities [CHE05; Rie+02] to specify lower and upper bounds on the number of selected sub-features. Feature attributes [BTR05; CBH11] may be utilized to constrain the valid feature combinations depending on quantitative measures, e.g., the combined cost, size, or energy consumption of features. Finally, cardinalities may also be attached to features, leading to multi-features [Cor+13c], i.e., features that can appear multiple times in a feature combination. All mentioned extensions are orthogonal and can be combined in the same feature model notation. Initially, the semantics of FODA-style feature diagrams was only intuitively defined which lead to ambiguities and raised questions regarding their expressiveness. This issue was later remedied by giving them a formal semantics [Sch+07]. Textual representations of feature diagrams, e.g., the *Feature Description Language* (FDL) [VK02] and the *Textual Variability Language* (TVL) [CBH11], were introduced to overcome the limitations of the graphical notation w.r.t. scalability and tool support. While feature diagrams and their textual representations are the de-facto standard for feature modeling, alternative representations have been proposed, e.g., propositional formulas [Bat05] and constraint-based characterizations [Jör+12].

The feature-oriented development process is separated into two distinct phases [CN02; Gri00], as depicted in Figure 3.2. In the *domain engineering* phase, the base functionality as well as the variability within the system family are identified which results in a feature model defining the set of features and valid feature combinations. Artifacts corresponding to those features are then created and implemented. In the *application engineering* phase, demands and requirements are collected to refine the feature model until a single suit-

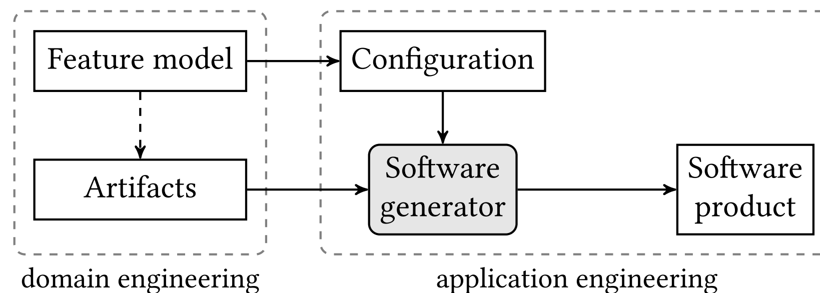


Figure 3.2: The feature-oriented development process

able configuration (i.e., feature combination) is found. Practical product-line approaches typically provide a software generator tool which automates the derivation of a concrete product from the artifact base for a given configuration. For this, several techniques have been proposed. Annotative approaches (e.g., using `#ifdef` directives) commonly use an off-the-shelf preprocessor for selecting relevant portions of code [Lie+10]. Generative approaches [CE00] as well as feature-oriented programming [AKL09; Bat04], delta-oriented programming [Sch+10], and aspect-oriented programming [Kic+97] allow for a modularization of feature-related functionality in feature modules, delta modules, and aspects, respectively.

Feature combinations are not necessarily static. In a *dynamic software product line*, the set of features, i.e., the configuration, can be changed by activating and deactivating features at runtime [Din+10; DS11; GH03]. This enables upgrades of already deployed systems. Moreover, dynamic features can be applied to model and implement self-adaptive systems where features are activated and deactivated in response to changes in the environment.

Features are usually not completely independent from each other. The notion of *feature interaction* refers to the emergent behavior that arises from the combination of two or more features that is not present when the features are applied in isolation [Ape+11; Ape+13a; PR01]. Such interactions are often intended and unavoidable to implement the required functionalities of a system in a modular fashion. However, *unintended* interactions may cause unforeseen side-effects that severely compromise the correctness of the system. Unintended feature interactions have been extensively studied in the context of telecommunication systems [Cal+03]. An example from this domain are the features “voice mail on busy” and “forward call on busy” which both try to take control of a second incoming call [Cal+03; PR01]. Prioritizing either one of them inadvertently breaks the other. Since the number of possible feature combinations is usually exponential in the number of features, the detection and prediction of feature interactions is challenging. Within dynamic feature-oriented systems, the problem becomes even more difficult, as the temporal order of feature activations and deactivations may have an impact on the occurring interactions. The extended notion of *quantitative feature interactions* refers to

interactions that have an influence on the non-functional properties of the system, such as performance [Sie+12].

Towards detecting feature interactions, several analysis techniques have been adapted to feature-oriented systems, including type checking, static analysis (e.g., control-flow and data-flow analysis), theorem proving, and model checking [Thü+14a]. To deal with the possibly large number of feature combinations, these approaches exploit the commonalities between the individual products, e.g., by utilizing symbolic representations, or make use of the compositional structure by applying compositional reasoning techniques. In the following, we will discuss the related work that applies model checking for the formal analysis of feature-oriented systems. FTS [Cla+10; Cla+13] are a variant of labeled transitions systems where transitions are annotated with feature combinations, i.e., FTS follow the annotative approach. With that, a single FTS can represent the behavior of a whole system family. Tool support for analyzing FTS is provided by SNIP [Cla+12; Cla+13]. Its input language fPromela extends the Promela language [Hol97] with feature annotations. Cordy et al. later added support for multi-features and feature attributes [Cor+13c]. Using SNIP, a system family can be verified against properties expressed in LTL. ProVeLines [Cor+13b] is a re-engineered implementation of SNIP and is itself a product line of verification tools. In addition to LTL model checking, it supports the verification of timed models and is able to check the behavioral equivalence of SPLs. While FTS can only represent static product lines, A-FTS [Cor+13a] allow for changes to the feature combination within transitions and are thus able to represent dynamic SPLs. Plath and Ryan [PR01] adapted the superimposition construct [Kat93] to the input language of the model checker NuSMV. In contrast to the annotative approach in fPromela, the resulting language allows for the definition of feature modules. In the superimposition approach, a feature module describes the changes it applies to a system upon composition. Plath and Ryan [PR01] only consider a product-based analysis, i.e., each system variant has to be constructed and analyzed one-by-one. A family-based analysis approach for fSMV (the feature-oriented extension of NuSMV's input language) has been proposed by Classen et al. [Cla+14]. They applied the *lifting* approach [PS08] to encode the feature combinations into the model's state space. A minor extension to NuSMV enables the extraction of the satisfying feature combinations after checking the system family against a CTL property. A similar approach based on lifting has been proposed in [Dim+15] for translating fPromela into the standard Promela language. The tool SPLverifier [Ape+11] utilizes FEATUREHOUSE [AKL09] for the composition of feature modules written in the C language. This approach also employs lifting which then allows the analysis of an SPL using the non-feature-aware model checker CPAChecker [BK11]. Lauenroth et al. [LPT09] propose I/O-automata with "must" and "may" transitions for representing system families and provide a model checking tool for checking them against CTL-like properties. Based on modal transition systems [Asi+11], the tool VMC [BMS12] enables the verification of SPLs against requirements expressed in a branching-time logic. While it

is not primarily an analysis tool, the FeatureIDE [Thü+14b] integrates the theorem prover KeY as well as the JPF-BDD model checker for the verification of product lines written in Java. All approaches mentioned up to this point only deal with non-probabilistic behavior and only allow for a functional analysis. The work by Ghezzi and Sharifloo [GS13] as well as the approach by Rodrigues et al. [Rod+15] apply parametric model checking for the analysis of SPLs. Here, the parameters of a parametric *discrete-time Markov chain* (DTMC) encode the variability within the system family. A similar approach for capturing variability in UML activity diagrams has been presented by Kowal et al. [KST14] where parametric *continuous-time Markov chains* (CTMCs) provide the basis for a family-based performance analysis. The tool QFLan [Van+18] provides a modeling language for defining feature-oriented systems with probabilistic behavior and possibly dynamic features. Here, models have a DTMC semantics and are translated into the input language of the MultiVeStA model checker for statistical model checking. In the input language of QFLan, feature guards are tied to actions instead of individual transitions (as it is the case for FTS) which severely limits the compositionality of feature modules. A compositional modeling framework for probabilistic feature-oriented systems has been introduced by Dubslaff et al. [DKB14] and extended in [DBK15]. Rather than relying on superimposition, the framework promotes parallel composition for combining feature modules. The changes of the configuration in a dynamic feature-oriented system are described using a special coordination component called feature controller. The semantics of a set of feature modules under a feature controller is given in terms of a single MDP that encompasses the behavior of the whole system family. Here, the initial states of the MDP correspond to the initial feature combinations. The parallel composition of feature modules can be mapped straightforwardly to the parallel composition of modules as implemented in the model checker PRISM. The framework in [DBK15] has been implemented in the tool ProFeat [Chr+18] which will be presented in detail in the following sections.

3.2 Describing system families: The ProFeat language

This section presents the ProFeat language which can be considered as an extension of the PRISM language tailored to model families of stochastic systems using feature-oriented constructs. Additionally, the ProFeat language provides support for metaprogramming to describe families of systems generically.

In the following, the ProFeat language will be illustrated using a simple producer-consumer running example as shown in Figure 3.3. The system comprises a single producer that creates jobs with workload sizes chosen according to a stochastic distribution and enqueues them into a FIFO buffer. The workers can take jobs from this buffer and process them. Each worker can only process a single job at once. The time it takes to process a job is determined by the job's workload size and the processing speed of the worker. Varying the buffer size, the number of workers, the workload distribution, and the processing

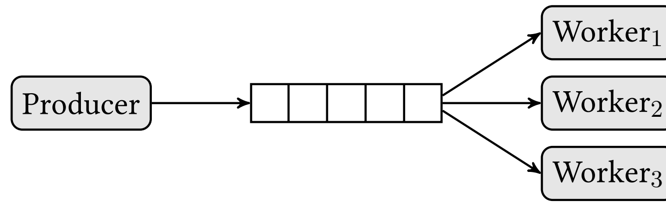


Figure 3.3: Running example: a producer-consumer system with a single producer, multiple consumers, and a buffer in between

speed of the workers gives rise to a family of systems. The accompanying feature diagram is shown in Figure 3.4.

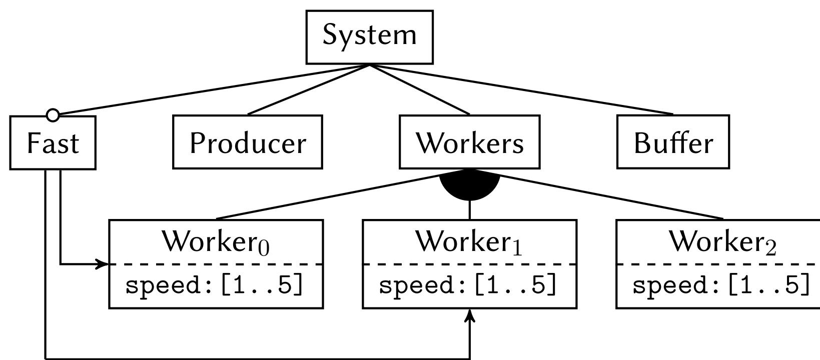


Figure 3.4: Feature diagram of the producer-consumer system

3.2.1 Feature-oriented language constructs

A ProFeat model consists of two distinct parts: a declaration of a feature model describing the feature combinations and a modular definition of the operational behavior of some or all features declared in the feature model. Optionally, a set of family parameters may be specified. For the definition of operational behavior, the ProFeat language adopts the guarded command language of PRISM and extends it with feature-specific concepts as presented in [DBK15]. By using constraints over the activity of the declared features, the behavior of features or the base system can depend on the current feature combination. Furthermore, dynamic feature-oriented systems can be specified by defining a special module called *feature controller* that is responsible for switching between different feature combinations. The controller module uses the same syntax as other modules, but can additionally activate and deactivate features as part of the update definition within commands.

```
root feature
  all of Producer, Buffer, Workers, optional Fast;
  constraint active(Fast) => active(Worker[0]) & active(Worker[1]);
  constraint Worker[0].speed + Worker[1].speed < 7;
endfeature

feature Workers
  some of Worker[3];
endfeature

feature Worker
  speed : [1..5];
endfeature
```

Listing 3.5: Excerpt of the feature model for the producer-consumer system

Declaration of the feature model

ProFeat provides a textual description language for declaring feature models that is inspired by the *Textual Variability Language* (TVL) [CBH11]. A part of the feature model declaration for the producer-consumer running example is shown in Listing 3.5. Features are declared within feature blocks enclosed by the keywords `feature` and `endfeature`. The `root feature` is a designated feature that represents the base system. In the example, the root feature corresponds to the System feature in Figure 3.4 and is decomposed into four sub-features. The `all of` decomposition (line 2) states that all listed sub-features must be part of the feature combination whenever their parent feature is selected or active. The Worker feature is decomposed using `some of` (line 8) which indicates that at least one of the sub-features must be selected. In addition to the `one of` operator (which requires exactly one of the sub-features to be active), the decomposition can also be given in terms of a cardinality `[n..k]` declaring that at least `n` and at most `k` sub-features must be active whenever the parent feature is active. Optional features are preceded by the `optional` keyword (line 2), stating that the feature may or may not be part of the feature combination, regardless of the decomposition operator, i.e., the `optional` keyword has priority over the decomposition operator.

ProFeat also supports *multi-features* [Cor+13c], i.e., features that can appear multiple times in a feature combination. The number of instances is given in brackets after the feature name. In the example, the Workers feature comprises three copies of the Worker feature (line 8). If a multi-feature is marked as `optional`, then each individual copy is optional. Besides multi-features, ProFeat also supports *feature attributes* [CBH11]. For instance, the Worker feature has the attribute speed which can take any integer value

from 1 to 5. Using multi-features and feature attributes, even large and complex product lines can be specified succinctly.

The inclusion of multi-features in the language necessitates a distinction between features and feature instances. In ProFeat, each feature instance is uniquely identified by a *fully qualified name* that corresponds to the path from the root feature to the feature instance in the feature hierarchy. The sub-features of a feature as well as its attributes are accessible using the familiar dot-notation. If a sub-feature is a multi-feature, the specific instance is referred to using the index-operator. For instance, the fully qualified name of the second Worker's `speed` attribute is `root.Workers.Worker[1].speed`. As long as the qualified name remains unambiguous, the prefix of the name can be dropped, e.g., `Worker[1].speed` is valid as well.

Analogous to feature diagrams, the textual feature model may also contain cross-tree constraints as well as constraints over the values of feature attributes. In the running example, the first constraint (line 3) in the root-feature block requires that the first two Worker instances must be active whenever the Fast feature is selected. The second constraint limits the combined speed of the first two Workers. If a constraint is preceded by the **`initial`** keyword, then it only needs to hold for the initial feature combination, but is not required to be satisfied after changes to the feature combination issued by the feature controller. Obviously, this distinction is only relevant if the model defines a controller and thus allows for dynamic feature switches.

Definition of feature behavior

The declaration of the feature model is strictly separated from the operational behavior of features which enables an easy reuse of feature models and behavior definitions. A feature can be "implemented" by one or more *feature modules* that are referenced using the **`modules`** keyword within the feature declaration. For instance, the Worker feature is implemented by the `Worker_impl` module, as shown in line 5 of the extended feature declaration in Listing 3.6. Note that ProFeat follows the approach of [DBK15] where feature modules are defined using the annotative approach and where the overall system behavior arises from the parallel composition of all feature modules. This in contrast to, e.g., fSMV where feature modules are defined by a superimposition construct [Kat93]. Superimposition is order-dependent, i.e., the feature module which is added last can not only change the behavior of the base system, but also that of all feature modules that have been added before it. This makes adding and removing features along with their respective feature modules rather challenging. Therefore, ProFeat employs an annotative approach using a parallel composition of feature modules.

Feature modules are defined using an extended version of PRISM's syntax for modules. A module comprises a set of local variables (line 9 in Listing 3.6) that define its local state space and a set of guarded commands (lines 11 and 12). A guarded command consists of an optional action label (in brackets), a guard over the variables of the model,

```
feature Worker
  speed : [1..5];

  block dequeue[id];
  modules Worker_impl;
endfeature

module Worker_impl
  t : [0..max_work_size] init 0;

  [working[id]] t > 0 -> (t' = max(0, t - speed));
  [dequeue[id]] t = 0 -> (t' = Buffer.cell[0]);
endmodule
```

Listing 3.6: Declaration and implementation of the Worker feature

and a stochastic update of the module's local variables. For instance, the command in line 11 expresses that the module processes remaining work units by subtracting its speed, but only if there are still work units available (`t > 0`). Modules can communicate by synchronizing over shared actions or by reading each other's local variables. Local variables can be accessed using the same dot-notation as described earlier for features and attributes. Within the local scope of a feature and its corresponding feature modules, one can refer to its own local variables and attributes without using a qualified name. For instance, the `speed` attribute of the Worker feature is referenced in line 11. The local variables of a feature module can always be read, even if the corresponding feature is not active. If a feature is deactivated, its local state is "frozen" and remains unchanged until the feature is reactivated. For defining behavior that depends on the current feature combination, ProFeat provides the **`active`** predicate that evaluates to `true` if the given feature is active in the current system state. By default, feature modules of inactive features do not block on synchronizing actions. Thus, with respect to synchronization, deactivating a feature has the same effect as removing its feature modules from the model. This is useful if the model is fully synchronous, i.e., if there is a global action that synchronizes the transitions of all modules. However, in some cases the non-blocking behavior is undesired. Then, it is crucial that the synchronization with actions of an inactive feature is prevented, i.e., blocked. In the producer-consumer example, taking a job out of the buffer is modeled using synchronization (line 12) where each Worker has its own individual `dequeue` action[2]. Thus, an inactive Worker must not synchronize

[2] In ProFeat, action labels can be indexed like arrays. This construct is utilized here to generate unique `working` and `dequeue` actions for each instance of the Worker feature. The **`id`** expression evaluates to the index of the feature instance.

```
1 controller
2   [] buffer_full & !active(Worker[2]) -> activate(Worker[2]);
3   [] buffer_low & active(Worker[2]) -> deactivate(Worker[2]);
4 endcontroller
```

Listing 3.7: Feature controller for the producer-consumer system

with the buffer, or else the job is lost. Therefore, its `dequeue` action is marked as blocking (line 4).

Definition of the feature controller

The *feature controller* is a special module that defines rules for the dynamic activation and deactivation of features. Listing 3.7 shows an example for the producer-consumer model. Essentially, a **`controller`** is a module (possibly with its own internal state) which can modify the feature combination using the **`activate`** and **`deactivate`** updates. In the example, the third Worker is activated to speed up processing whenever the buffer is full. Conversely, if the buffer is almost empty, the now superfluous Worker is deactivated, e.g., to save energy. The definition of the controller is optional. If no controller is given, the feature combination is static. Feature modules can synchronize with the controller upon activation or deactivation using the special **`activate`** and **`deactivate`** actions which allows them to react to the activation or deactivation of their associated feature to, e.g., reset local variables. Furthermore, a feature module may also block its deactivation which is useful for deferring the deactivation to a later point. For instance, by adding the following line to the `worker_impl` module, the Worker cannot be deactivated until it has finished processing its current job:

```
[deactivate] t = 0 -> true;
```

3.2.2 Parametrization

Model families can also arise from system parameters. For instance, in the producer-consumer example the size of the FIFO buffer can be considered as such a parameter. System parameters are defined within a **`family`** block, as shown in Listing 3.8. Similar to feature attributes, system parameters can be constrained as well (line 3). It is possible to combine a family declaration with a feature model which results in a system family that is defined both by system parameters and all valid initial feature combinations.

System parameters can be used anywhere in the model including, but not limited to, guards and probabilities. In contrast to feature attributes, system parameters are treated as constants within each instance of a family. This has an important consequence: parameters can also be used to specify the range of variables, the size of arrays, and even

```
1 family
2   buffer_size : [1..8];
3   initial constraint buffer_size != 5;
4 endfamily
```

Listing 3.8: Parametrization of the producer-consumer model

the number of multi-feature instances. Thus, system parameters can directly influence the structure and the state space of the system.

3.2.3 Metaprogramming language extensions

ProFeat provides several extensions to the PRISM language that enable a generic definition of models. This is especially important for modeling system families as it allows us to represent the behavior of all system instances using a single parametrized definition. Besides Boolean and integer variables, ProFeat supports (one dimensional) arrays. Additionally, the language offers several metaprogramming constructs that are commonly provided by template languages. Using `for` loops, sequences of commands, probability distributions, variables updates, and expressions can be generated. A common use case are families where the family instances differ in their structure, e.g., in the buffer size or the number of multi-feature instances. If a `for` loop is used in an expression, its body must contain the placeholder "..." exactly once. Intuitively, in the iteration i this placeholder is replaced with the resulting expression of iteration $i + 1$. For instance, the following expression sums up the first n elements of the array `arr`:

```
for i in [0..n-1] { arr[i] + ... }
```

If $n = 4$, this expression is expanded to:

```
arr[0] + arr[1] + arr[2] + arr[3]
```

Feature templates and module templates

Both `feature` blocks and feature modules can be instantiated multiple times. Each time a feature is referenced in a decomposition, a new instance of that feature and all its associated feature modules is created. Thus, feature declarations and feature modules can be regarded as reusable templates. These templates can be parametrized which in turn enables a parametrization of guards, probabilities, and costs.

An example is the feature module implementing the Buffer in the producer-consumer model. It is parametrized over the capacity of the buffer, as shown in line 5 of Listing 3.9. The actual buffer size is determined upon instantiation of the feature module (line 2). In the given example, the buffer size is defined as a system parameter `buffer_size`

```
1  feature Buffer
2    modules fifo(buffer_size);
3  endfeature
4
5  module fifo(capacity)
6    cell : array [0..capacity - 1] of [-1..max_work_size] init -1;
7
8    for w in [0..2]
9      [dequeue[w]] cell[0] != -1 ->
10       (cell[capacity-1]' = -1) &
11       for i in [0..capacity-2] (cell[i]' = cell[i+1]) endfor;
12   endfor
13
14   // ...
15 endmodule
```

Listing 3.9: A FIFO buffer implementation parametrized over the capacity

ranging over a finite set of possible values. The `for` loop spanning lines 8 to 12 generates a command for each worker labeled with the dequeue action. The inner loop (line 11) shifts the buffer contents when the first element is taken out.

3.2.4 Property specifications

For the definition of specifications and queries, ProFeat reuses PRISM's property specification language which is based on *probabilistic computation tree logic* (PCTL) [AHK03; BA95; BK98]. Within properties, ProFeat's full range of language extensions may be used. We consider three example properties of the producer-consumer model. The following property states that even in the worst case almost surely the buffer is not filled more than 75%:

$$\mathrm{Pr}^{\min}\big(\Box\,(\mathsf{level} < 0.75)\big) = 1$$

As a second example, we ask for the minimal probability that at some point Worker_2 is active which can be expressed by:

$$\mathrm{Pr}^{\min}\big(\Diamond\,(\text{“}\mathsf{Worker}_2\text{ is active”})\big)$$

ProFeat models may be annotated with costs and rewards for reasoning about quantitative measures. In contrast to the PRISM language where reward structures are defined globally, in ProFeat rewards are defined locally as part of feature declarations. Not only does this further facilitate a modularization of the model, it also has practical implications.

```
1 feature Worker
2   rewards "energy"
3     active(this) & t > 0 : 1;
4     [activate] true : 5;
5   endrewards
6 endfeature
```

Listing 3.10: Definition of a Worker's energy consumption

As part of a feature declaration, rewards may also be parametrized using ProFeat's feature template construct. ProFeat extends the reward syntax of PRISM in two ways. First, the `active` function may also be used within reward definitions, allowing for rewards to depend on the feature combination. Second, rewards can be attached to feature switches by using the special `activate` and `deactivate` actions, enabling quantitative reasoning about dynamic feature-oriented systems. Listing 3.10 shows an extended declaration of the Worker feature encompassing a reward structure for its energy consumption. In line 3, energy costs of 1 are specified for all states where the Worker is active and currently processes a job. The `this` keyword refers to the feature instance for which the feature declaration has been instantiated. All feature switches that activate a Worker feature are annotated with a cost of 5 (line 4). The third example property specifies that even in the worst case the maximal expected energy consumption of the producer-consumer system does not exceed a given threshold. Here, goal is an atomic proposition which holds in those states where all jobs have been processed completely.

$$\mathrm{Ex}^{\max}(\text{“accumulated energy until reaching goal”}) \leq \text{threshold}$$

Listing 3.11 shows all three properties (lines 4, 5, and 13) in the extended specification language. Expressions containing ProFeat-specific language constructs must be enclosed in `${` and `}`. For instance, in the definition of the atomic proposition `goal` (lines 8–11), a `for` loop is used to iterate over all Worker instances. The `active` function can be used in the same way as in a ProFeat model.

3.2.5 Semantics

A ProFeat model may begin with a keyword determining the type of model that is described, i.e, either a DTMC (keyword `dtmc`), a CTMC (keyword `ctmc`), or an MDP (keyword `mdp` which is the default if no model type is given). The semantics of the ProFeat model then is a family of DTMCs, CTMCs, or MDPs respectively, and can be obtained straightforwardly from the following three ingredients. First, ProFeat applies the framework presented in [DBK15] which provides a formal semantics for feature modules under a feature controller. Second, the semantics of the feature model follows that of

```
1  formula level = ${ (for i in [0..buffer_size-1]
2                       (cell[i] > 0 ? 1 : 0) + ...
3                     endfor) / buffer_size } ;
4  Pmin>=1 [ G (level < 0.75) ];
5  Pmin=? [ F ( ${ active(Worker[2])) } ];
6
7  const threshold = 20;
8  label "goal" = ${ (counter = 0) &
9                   for w in [0..2]
10                    (active(Worker[w])) => Worker[w].t = 0 & ...
11                  endfor } ;
12
13 R{"energy"}max<=threshold [ F "goal" ];
```

Listing 3.11: Property specifications utilizing ProFeat's language extensions

TVL [CBH11] extended with multi-features as proposed in [Cor+13c]. Finally, ProFeat uses a translational approach towards PRISM models whose guarded command language has a well-defined semantics [KNP11]. The translation of a ProFeat model into a PRISM model is detailed in the following section.

3.3 Implementation

This section gives an overview of the ProFeat tool and provides notable implementation details. First, we discuss the general workflow of using ProFeat for the analysis of system families described in the ProFeat language. As depicted in Figure 3.12, ProFeat applies a translational approach utilizing the probabilistic model checker PRISM for carrying out the analysis. ProFeat models (and properties) are translated into the input language of PRISM. In a post-processing step, the analysis results produced by PRISM are collected and transformed such that they refer to the original ProFeat model. While those three steps are usually performed in succession, they may also be executed separately to, e.g., apply some further transformations on the generated PRISM models or to explore the generated model using a simulator.

ProFeat provides two different analysis approaches: *one-by-one* and *all-in-one*, also called product-based analysis and family-based analysis in the context of software product lines [Thü+14a]. In case of a one-by-one analysis, ProFeat generates a separate PRISM model for each member of the system family which, consequently, are analyzed separately by invoking PRISM for each instance. In the all-in-one approach, a single model representing the whole family is generated. The analysis of this model using a single run of the model checker yields the results for all family members at once.

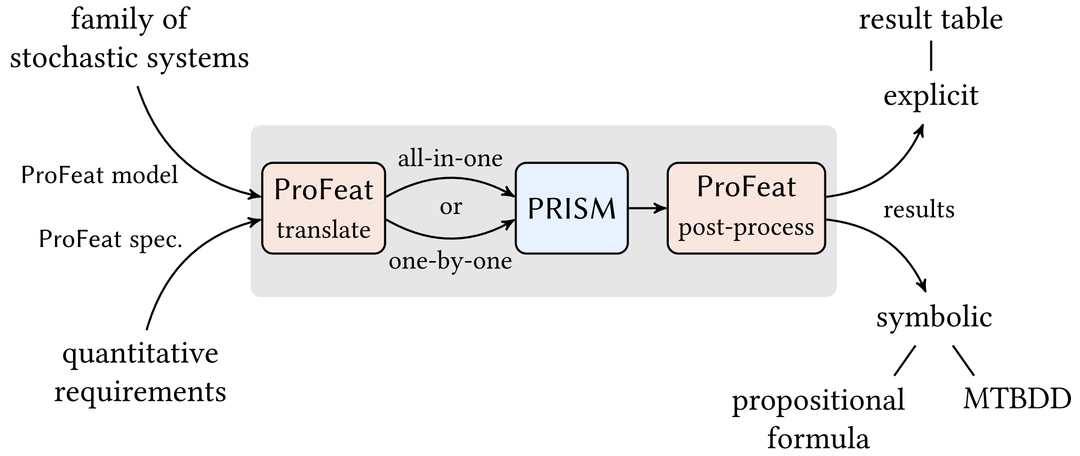


Figure 3.12: ProFeat workflow overview

Clearly, the commonalities between the family members may cause a lot of redundancy in the state space of the family model. ProFeat relies on PRISM's symbolic engine which internally uses BDDs, a universal data structure for representing Boolean functions, to compactly store the model. A BDD [Ake78; Lee59] is a directed, acyclic graph with a single root node and consists of decision nodes as well as terminal nodes representing the constants true and false. Each decision node is labeled with a Boolean variable and has exactly two outgoing edges corresponding to setting the variable to true or false, respectively. Within ordered BDDs [Bry86], the same ordering of variables is used consistently along all paths from the root to the terminal nodes. The graph structure of an ordered BDD is obtained by reducing the decision tree of a Boolean function (using the fixed variable ordering) by merging isomorphic subgraphs and eliminating any terminal nodes with the same value as well as any decision node where both outgoing edges target isomorphic subgraphs. Removing redundancies and exploiting shared structures in the BDD yields the potential of compactly representing Boolean functions. When using BDDs to represent family models, this means that behavior that is shared between instances of the family is potentially present only once. Since PRISM is a probabilistic model checker, it actually uses *multi-terminal binary decision diagrams* (MTBDDs) to represent the model. MTBDDs [Bah+93; Cla+93] are a variant of BDDs where terminal nodes can represent any real value instead of just the values true and false. The memory consumption and the analysis performance crucially depend on the variable ordering of the MTBDD. We can exploit automated reordering techniques [Kle+18] to minimize the size of the MTBDD representation which potentially speeds up the analysis.

It may seem that the all-in-one approach is clearly superior to the naive one-by-one approach, however, this is not necessarily the case in practice. Since the family model must encompass the behavior of all family members, it is usually much larger than any of the models for a single family member. Thus, if the family model is too large to fit in

memory, a one-by-one analysis might still be feasible. Moreover, the one-by-one approach can be trivially parallelized. If enough computational resources are available, the analysis of the whole family takes just as long as the analysis of the largest family member. In conclusion, it largely depends on the model and the available resources which approach is favorable. Therefore, ProFeat lets the user switch between the all-in-one and one-by-one approaches without requiring any modifications of the ProFeat model. The translation of properties does not depend on the chosen analysis approach and merely involves a syntactical transformation of ProFeat language constructs into their PRISM-language equivalent.

The main advantage of the translational approach is that the full range of PRISM's support for quantitative analysis can also be employed for feature-oriented models. Furthermore, any improvements to the PRISM implementation also benefit ProFeat. There is also the potential for using a different model checking tool that supports PRISM's input language, e.g., Storm [Hen+20].

The post-processing step transforms the analysis results returned by PRISM into a list of results for each feature combination and each possible combination of parameter values. In case a one-by-one analysis was chosen, ProFeat automatically collects the results from each analysis run. Thus, the representation of the final results is independent of the chosen analysis approach. The *explicit* representation of results in the form of a table is usually sufficient for small system families or if only a few configurations are of interest. However, if the system contains many features, the exponential blow-up of the number of feature combinations leads to numerous entries in the result table. This makes it difficult to draw general conclusions about the whole system family. In order to give the user a better understanding of the analysis results, ProFeat can additionally provide *symbolic* result representations. Similar to the analysis, where the symbolic representation is often more compact than the explicit representation, a symbolic representation of the results is often smaller than the list of feature combinations. Two different representations are supported. First, a propositional formula over features representing the set of feature combinations satisfying some qualitative property can be generated (similar to the output returned by the feature-aware model checker ProVeLines [Cor+13b]). Note that this includes quantitative properties where some threshold must be met. Second, ProFeat can produce a BDD where each inner node represents a decision whether the corresponding feature is enabled or not, and each terminal node indicates whether the (qualitative) property is satisfied or not. In case of a quantitative analysis, an MTBDD is generated instead where each terminal node corresponds to a distinct analysis result. ProFeat may additionally apply rounding to the analysis result which potentially reduces the number of terminal nodes at the cost of precision. The generated MTBDD may be used within the application engineering phase to guide the feature selection depending on quantitative measures.

In the following, the implementation of the translation as well as the post-processing of analysis results will be discussed in detail.

3.3.1 Translation of ProFeat models

The translation of a set of feature modules under a feature controller into a PRISM model is based on the compositional modeling framework for feature-oriented systems presented in [DBK15] which maps the composition of feature modules to the parallel composition of PRISM modules. The translation of ProFeat specifications into PRISM specifications is purely syntactical and involves the translation of ProFeat expressions into PRISM expressions within properties. In the following, notable steps of the translation are highlighted and illustrated using the producer-consumer running example introduced in Section 3.2.

Encoding of feature combinations

Since PRISM is not a feature-aware model checker, we apply the *lifting* approach [PS08] to encode the set of feature combinations into the state space of the model. In the ProFeat language, the read and write access to the feature combination is only possible via the `active` function and the `activate`/`deactivate` updates, respectively. This abstraction allows us to choose an internal representation of the feature combinations that best suits our needs. The basic idea is to introduce a new Boolean variable for each feature in the feature model that is set to `true` if the feature is part of the feature combination and set to `false` if not. However, some features may be mandatory, i.e., they must be part of every feature combination, and thus their associated variable would always be set to `true`. Furthermore, the inclusion of some feature may solely depend on the inclusion of their parent feature. In order to reduce the number of variables for representing the feature combination, only one variable per non-mandatory *atomic set* is generated. An atomic set consists of features that can be treated as a unit, as they always appear (or do not appear) together in feature combinations [Seg08]. For instance, in the translation of the producer-consumer system (see Figure 3.4), a variable for the Fast feature and one variable for each Worker instance is generated. ProFeat does not actually use Boolean variables, but rather integer variables with a range of `[0..1]`, as this enables summing up feature variables in the PRISM language and thus simplifies the handling of cardinality constraints. With this representation in place, the translation of the `active` function is straightforward. If the atomic set associated to the feature is mandatory, then the expression is simply replaced by `true`. Otherwise, the corresponding feature variable of the atomic set is inserted in place of the call. Analogously, the `activate` and `deactivate` updates assign `0` or `1` to the associated variable, respectively.

Translation of feature modules

Each feature module instance is translated into a single PRISM module. Consider Listings 3.13a and 3.13b which show the feature module of the Worker feature and its translation, respectively. Note that only the translated module for the first Worker is shown since the other instances are nearly identical. As all variable names are global in the PRISM language, the variable names are prefixed with the feature instance name to make them unique and to prevent name clashes in the translated model. Here, `w_0` is the internal name of the first Worker feature instance where `0` is the index of the multi-feature instance. The guard of each command is extended with the corresponding feature variable `w_0_act` such that the commands can only be executed when the associated feature is part of the feature combination. However, it must be ensured that an inactive feature does not block its actions, i.e., deactivating the feature should have the same effect as removing it. This is achieved by inserting transitions for each non-blocking action that can only be taken if the feature is not active (line 9 in Listing 3.13b). This command is not generated in case the action has been marked as blocking in the feature declaration. In the example, the dequeue action has been marked as blocking, thus no additional command is inserted.

```
1  module Worker_impl
2    t : [0..max_work] init 0;
3
4    [] t > 0 ->
5      (t' = max(0, t - speed));
6    [dequeue[id]] t = 0 ->
7        (t' = Buffer.cell[0]);
8    [cancel] true -> (t' = 0);
9
10 endmodule
```

(a) Worker in ProFeat model

```
module w_0_Worker_impl
  w_0_t : [0..max_work];

  [] w_0_act & w_0_t > 0 ->
    (w_0_t' = max(0,w_0_t - speed));
  [dequeue_0] w_0_act & w_0_t = 0 ->
    (w_0_t' = b_cell_0);
  [cancel] w_0_act -> (w_0_t' = 0);
  [cancel] !w_0_act -> true;
endmodule
```

(b) Worker 0 in PRISM model

Figure 3.13: Feature module of a Worker and its translation

Translation of the feature controller

The feature controller is translated into a single PRISM module as well. Note that the feature variables that indicate whether an atomic set of features is active or not are local variables of the translated controller module. This way, the controller is able to update the configuration also in action-labeled commands[3]. The translation must ensure that activating and deactivating features does not lead to an invalid feature combination. Consider the update in line 4 of Listing 3.14. According to the feature model (see Figure 3.4),

[3] In the PRISM language, updates to global variables are only allowed if the command does not synchronize with any other module.

```
1 controller
2   [] buffer_full & !active(Worker[1]) -> activate(Worker[1]);
3   [] buffer_full & !active(Worker[2]) -> activate(Worker[2]);
4   [] buffer_low   -> deactivate(Worker[2]);
5   [] buffer_empty -> deactivate(Worker[1]) & deactivate(Worker[2]);
6 endcontroller
```

Listing 3.14: Feature controller of the producer-consumer system

at least one Worker feature must be active at all times. Thus, this command may only be executed if at least one other Worker feature is active in the current configuration, otherwise it should block. This is achieved by extending the guard of the translated command as shown in line 2 of Listing 3.15. The guard is synthesized as follows. First, all constraints of the feature model (including those given by the decomposition of features) involving the updated feature are collected (in this example only the `some of` decomposition of the Workers feature, implicitly stating that at least one and at most 3 Worker features may be active). Then, the corresponding feature variables are replaced by their updated value. In this case, Worker 2 should be deactivated, thus the feature variable `w_2` is replaced with `0`. Then, the resulting expression only evaluates to `true` if the updated feature combination is valid. Hence, the command can only be executed if the feature combination is still valid in the next state.

```
1 [w_2_deactivate] buffer_low &
2   (1 <= w_0 + w_1 + 0) & (w_0 + w_1 + 0 <= 3) ->
3   (Worker_2' = 0);
```

Listing 3.15: Translation of the third feature controller command

Another aspect of the translation concerns the synchronization between the feature controller and the feature modules. Remind that a feature module can synchronize with the controller when its associated feature is activated or deactivated using the `activate` or `deactivate` actions, respectively. Consider again the feature controller in Listing 3.14, where in line 4 the controller implicitly synchronizes with the feature module of Worker 2, shown in Listing 3.16a. To realize this synchronization in the PRISM model, ProFeat automatically generates action labels for feature activation and deactivation, as shown in line 6 of Listing 3.16b (action `w_2_deactivate`). The translated controller command is then labeled with the action `w_2_deactivate` as well. The last controller command in Listing 3.14 deactivates `Worker 1` and `Worker 2` at once, thus it also must synchronize with both corresponding feature modules. However, the PRISM language only allows for at most one action label per command. Thus, both action labels are merged into a

single action label as part of the translation. However, this solution requires special care when translating feature modules. First, the action labels of all controller commands that deactivate Worker 2 are collected (lines 4 and 5 in Listing 3.14). Then, the command in the feature module is duplicated for each of those collected actions and labeled accordingly, as shown in Listing 3.16b. This translation realizes the synchronization between the feature controller and the feature modules even in case of multiple simultaneous activations and deactivations of features.

```
1 module Worker_impl
2   t : [0..max_work] init 0;
3
4   [deactivate] t = 0 -> true;
5
6
7
8 endmodule
```

(a) Worker in ProFeat model

```
module w_2_Worker_impl
  w_2_t : [0..max_work];

  [w_1_deactivate_w_2_deactivate]
    w_2_act & w_2_t = 0 -> true;
  [w_2_deactivate]
    w_2_act & w_2_t = 0 -> true;
endmodule
```

(b) Worker 2 in PRISM model

Figure 3.16: Translation of the synchronization between a feature module and the controller

All-in-one and one-by-one translation

If the user chooses the one-by-one analysis, a separate model for each possible set of initial values for system parameters and for each initial feature combination is created. The system parameters are constant for each instance and thus are inserted as constants in the translated models. However, the feature variables are still generated as variables, as the feature combination may be changed by the feature controller.

In case of an all-in-one analysis, ProFeat generates a single PRISM model with multiple initial states, one for each member of the system family. Since the ProFeat language allows array sizes, the number of multi-feature instances, and variable bounds to be defined in terms of system parameters, the size of these structures depends on the initial state and is no longer statically known at the time of translation. For instance, in the producer-consumer model, the buffer size as well as the number of workers may be defined in terms of system parameters. Clearly, the family model must accommodate every member of the system family, thus ProFeat generates arrays with the largest possible size, generates the maximal number of multi-feature instances, and creates variables with the least lower bounds and largest possible upper bounds. The maximal size of these structures and variables can be computed from the possible values of the system parameters which are known at translation time. The need for instantiating all structures with their greatest possible size is the main reason that the family model is often substantially larger than (most of) the models for the family members.

3.3.2 Post-processing of analysis results

As a consequence of the translational approach, the analysis results are produced by the employed analysis tool. Therefore, the results actually refer to the translated PRISM model rather than the ProFeat model, i.e., variable names will not appear as written in the ProFeat model. Furthermore, since PRISM is not a feature-aware tool, feature variables are not easily distinguishable from other variables. However, the main issue is that the results as produced by PRISM in a family-based analysis are hard to read which makes their interpretation challenging. As a first step of the post-processing, variable names and feature names are rewritten such that they match their definition in the original ProFeat model. Furthermore, feature combinations are represented as a set of feature names which further increases the readability.

Symbolic representation of analysis results

As ProFeat relies on standard tools for the analysis, the symbolic representation of the analysis results is not directly exported by the employed model checking tool. Thus, the propositional formula or the BDD representing the satisfying or violating feature combinations is computed from the list of results returned by PRISM. Generating the symbolic representations directly from the result table has a subtle side effect. Since only results for the valid feature combinations are returned by PRISM, the symbolic representation not only encodes the set of feature combinations satisfying the property, but also the set of valid feature combinations. This often makes the symbolic representation unnecessarily large, as it encodes information that is already provided by the feature model. For this reason, we apply techniques for *presence-condition simplification* [Rhe+15] and extend them to MTBDDs.

To generate a propositional formula representing the feature combinations that satisfy some qualitative property, ProFeat proceeds as follows. The set of satisfying feature combinations directly corresponds to a formula in *canonical disjunctive normal form* (CDNF) where the literals are features. In order to minimize this formula, the Quine-McCluskey algorithm [McC56] is applied. As mentioned, this formula also encodes the feature model, but we are only interested in the constraints that must hold in *addition* to the feature model. Formally, given a propositional formula Φ for the feature model, we need to compute an additional constraint Ψ such that all feature combinations of the restricted feature model $\Phi' = \Phi \wedge \Psi$ satisfy the property. This can be achieved by applying the Quine-McCluskey algorithm as described and additionally treating all invalid feature combinations as "don't care" terms which then gives the algorithm more opportunities for minimization.

The BDD or MTBDD representation of the analysis results is built by successively considering each line of the result table and adding its corresponding path to the BDD. As an example, we consider the simple model shown in Listing 3.17 and ask for the

```
1 root feature
2   all of optional x, optional y;
3   constraint active(x) => active(y);
4   modules base_impl;
5 endfeature
6
7 feature x modules x_impl; endfeature
8 feature y modules y_impl; endfeature
9
10 module base_impl
11   state : [0..1] init 0;
12   [tick] !(x.state = 1) & !(y.state = 1) ->
13       1/3: (state' = 0) + 2/3: (state' = 1);
14 endmodule
15
16 module x_impl
17   state : [0..1] init 0;
18   [] state = 0 -> 1/2: (state' = 0) + 1/2: (state' = 1);
19 endmodule
20
21 module y_impl
22   state : [0..1] init 0;
23   [tick] !(x.state = 1) -> 1/4:(state' = 0) + 3/4: (state' = 1);
24 endmodule
```

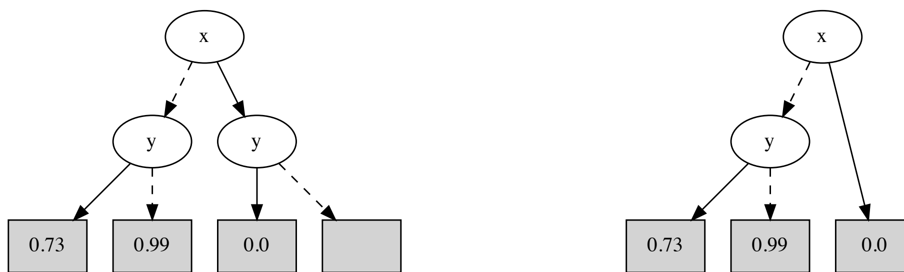
Listing 3.17: Simple feature-oriented model in ProFeat

```
Final result: [0.0,0.9876543209876543]
Results for initial configurations:
    (x, y)=0.0
    (y)=0.73
    ()=0.99
```

Listing 3.18: Output of analyzing the simple feature-oriented model with ProFeat

probability of reaching a state where `root.state = 1` holds. The result table is shown in Listing 3.18 and the corresponding MTBDD representation in Figure 3.19a. The variables in the MTBDD (indicated by ellipses) correspond to features. An outgoing solid line denotes that the respective feature is active, and a dashed line denotes inactive features. The terminal nodes correspond to the analysis results. Then, a path from the root node of the MTBDD to a terminal node stands for all feature combinations that share an analysis result. An additional terminal node (drawn as an empty box) stands for invalid feature combinations for which there is no analysis result. This is needed since the generated MTBDD still encodes the feature model. In order to remove any nodes from the MTBDD where the decision is already dictated by the feature model, we proceed as follows. First, the terminal node representing the invalid feature combinations is removed, together with all incoming edges. Now the MTBDD will contain inner nodes that only have one outgoing edge. Intuitively, this means that no actual decision is to be made in these nodes. The correct decision, i.e., the one that will not lead to an invalid feature combination, can be made by considering the feature model. Therefore, these nodes will be removed as well, again with all incoming edges. This procedure proceeds in a bottom-up manner until no more nodes can be removed. The resulting MTBDD for the example is shown in Figure 3.19b. Consider the path that starts with selecting the feature x in the root node. In the initial MTBDD, there is now a choice to include or not include feature y. However, from the feature model we already know that feature y must be active if feature x is active (line 3 in Listing 3.17). Therefore, this decision has been removed in Figure 3.19b.

The generated BDD representations of the results may still be quite large. Therefore, ProFeat provides a further reduction mechanism utilizing the *sifting* algorithm [Rud93]. The algorithm reorders the BDD variables in an attempt to find an equivalent BDD with fewer nodes. This typically yields an order where the features with the greatest impact on the analysis result are considered first. Thus, the relative “importance” of a feature w.r.t. the considered query can be easily extracted. In general, the height of the BDD scales linearly with the number of features. However, the number of BDD nodes largely depends on the structure of the analysis results and the variable ordering found by the sifting algorithm.



(a) Including feature model, rounded to 2 decimals (empty box represents an invalid feature combination)

(b) Without feature model, rounded to 2 decimals

Figure 3.19: MTBDD representations of analysis results

4 Case studies and application areas

In this chapter, the practical applicability of ProFeat for analyzing families of stochastic systems defined in terms of features is demonstrated by means of several case studies. All case studies have been conducted using ProFeat for translating the models and PRISM for carrying out the quantitative analysis. PRISM provides different model checking *engines* [KNP02]. The Explicit engine uses a graph-like data structure for representing models where each state (and its transitions) are stored individually in memory. The symbolic MTBDD engine uses MTBDDs for representing models, the Sparse engine uses sparse matrices, and the Hybrid engine uses MTBDDs for representing the model, but an explicit representation of vectors in the numerical analysis. Since ProFeat relies on a symbolic representation for removing redundancies in the family model representations, the Explicit engine has not been considered in the case studies. A performance comparison of the other three engines is part of Section 4.1.

Experiment setup. The experiments were carried out on a machine with two 8-core Intel Xeon E5-2680 CPUs running at 2.7 GHz with enabled Hyper-threading and equipped with 384 GB of RAM. The parallel execution of the one-by-one analysis runs was restricted to a maximum of 32 parallel processes.

Outline. The first set of case studies (Section 4.1) focuses on the comparison of the all-in-one and one-by-one analysis approaches, highlighting the approaches' suitability for different kinds of models and scenarios. Moreover, these case studies show the usefulness of the feature concept for defining a family of models, illustrating the benefit of using ProFeat for comparative studies, variant selection, and parameter optimization. The case studies in Section 4.2 show that ProFeat supports the analysis of stochastic SPLs, both static and dynamic. The body-sensor-network product line [Rod+15] furthermore illustrates how the symbolic result representation generated by ProFeat can help to gain insights on the impact of individual features on certain quantitative measures. The final two case studies in Section 4.3 show that the feature concept also has useful applications beyond the definition of SPLs, e.g., for capturing dynamic adaptations.

The results presented in this chapter have been published in [Chr+18], except for the last case study in Section 4.3.2 which is part of the publication [Wei+17].

4.1 Comparing family-based and product-based analysis

This section presents a set of case studies that have been conducted to compare the *all-in-one* (family-based) and *one-by-one* (product-based) analysis approaches, both for models defined in terms of *features* and for *parametrized* models. For the one-by-one analysis, we also consider a parallel execution of the analysis runs. To compare the analysis approaches for a system family defined in terms of features, the first case study examines variations of the producer-consumer system introduced in Section 3.2. The second set of case studies consists of several models defined in terms of system parameters which have been converted into ProFeat models to enable an all-in-one analysis.

4.1.1 Analysis of feature-oriented systems

In the following, the producer-consumer model (see Section 3.2) will serve as a base model for different variants and their analysis. Within the base model, which is a dynamic feature-oriented system, the feature controller can activate and deactivate Worker features, increase or decrease the size of the buffer, and increase or decrease the processing speed of each Worker. In order to guarantee fairness between the system's actions and the controller actions, an additional control module has been added. After a reconfiguration, this module blocks any controller actions until the producer has inserted at least one new job into the buffer. For the quantitative analysis, the goal is to process a certain number of jobs. The model has been extended with an additional module responsible for counting the finished jobs. The following three variants of this model and their corresponding analysis queries were considered:

Best buffer size. In this variant, the buffer size is static, i.e., not changed by the controller. Here, we ask for the buffer size that incurs minimal storage costs for processing all jobs. The storage costs depend on the number of filled buffer cells and also scale linearly with the buffer size, i.e., the larger the buffer, the higher the storage costs.

Best set of workers. This system is a static feature-oriented system where the variant space comprises all possible nonempty subsets of workers. Thus, for n different workers the system family has $2^n - 1$ members. We assume a heterogeneous set of workers where each worker has a different energy-efficiency. The goal is to find a combination of workers for which the expected energy required to finish all jobs is minimal.

Workload distributions. Here, we consider the fully dynamic base model as described above where the system variants arise from different workload distributions. The size of the system family is given by the number of possible distributions. The task is to find the distribution where the expected energy to finish all jobs is minimal.

Figure 4.1 shows the number of MTBDD nodes used by the representation of the three model variants for different buffer sizes, number of workers, and number of distributions, respectively. The number of nodes shown for the one-by-one approach is the sum of nodes for representing all individual family members. For all three models the number of nodes for the all-in-one approach is significantly smaller than the sum of nodes for the individual family members which indicates that some behavior is shared between the family members.

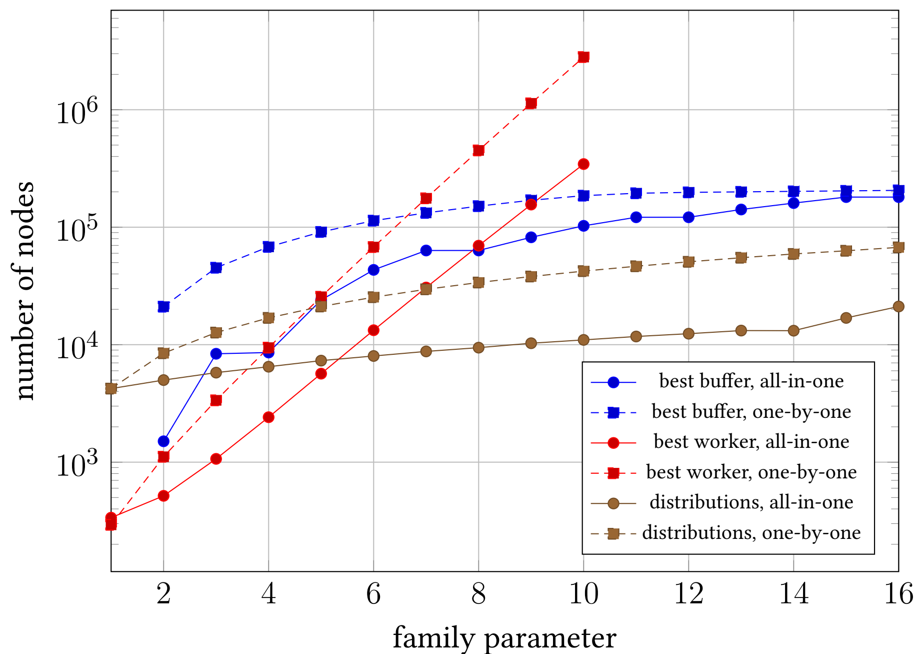


Figure 4.1: Number of MTBDD nodes for the producer-consumer models

The quantitative analysis described above has been carried out using both the MTBDD engine and the Sparse engine of PRISM. In general, the Sparse engine performed better than the MTBDD engine which is often the case for expectation queries (this is also the case for non-family models). The analysis times for finding the best buffer size are shown in Figure 4.2. Here, the all-in-one approach is only superior up to a maximal buffer size of 11 when using the MTBDD engine and 10 when using the Sparse engine. The structurally different buffer modules of the individual family members do not exhibit enough shared behavior that would benefit an all-in-one analysis. In case the number of family instances is exponential in the family parameter, as in the model for finding the best set of workers (see Figure 4.3), the all-in-one approach outperforms the one-by-one analysis and can even rival the parallel computation while using much fewer resources. For the third model variant, where different workload distributions were analyzed, the one-by-one and all-in-one approaches show asymptotically similar performance. Note that in this model the family instances only differ in the transition probabilities within the Producer

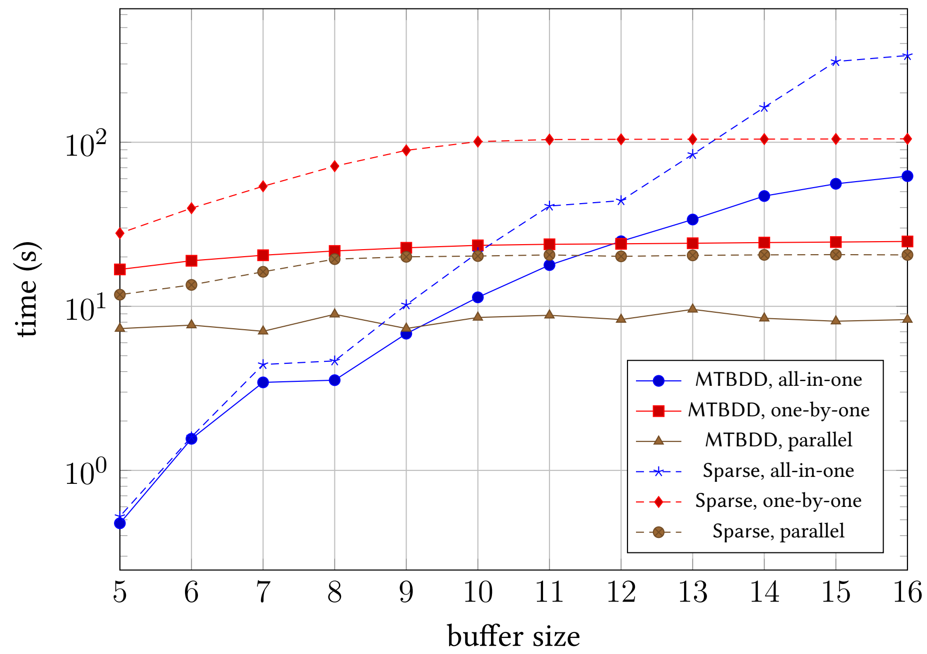


Figure 4.2: Analysis time of the producer-consumer model for an increasing buffer size

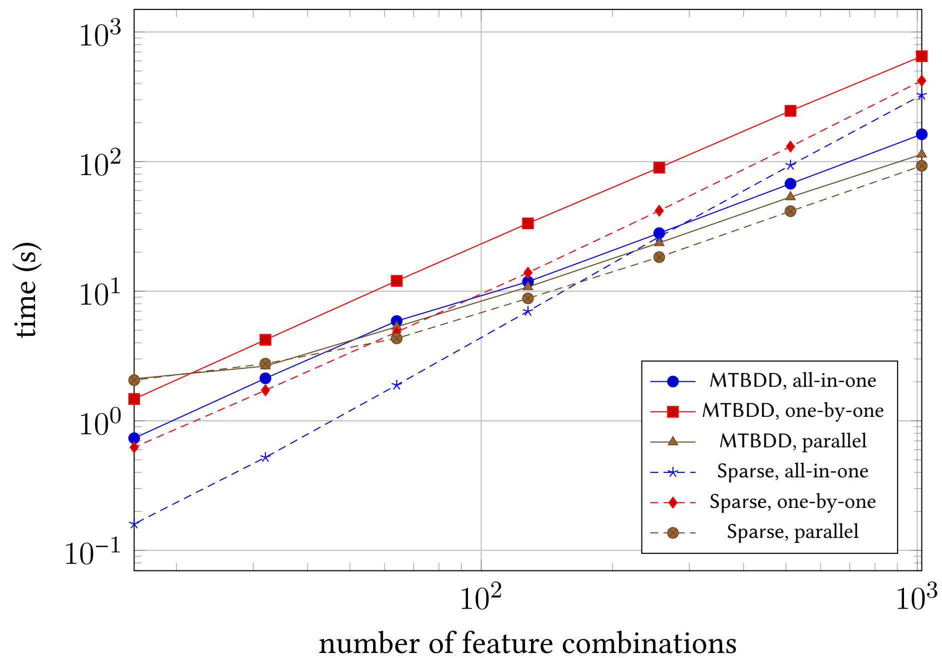


Figure 4.3: Analysis time of the producer-consumer model for an increasing number of workers

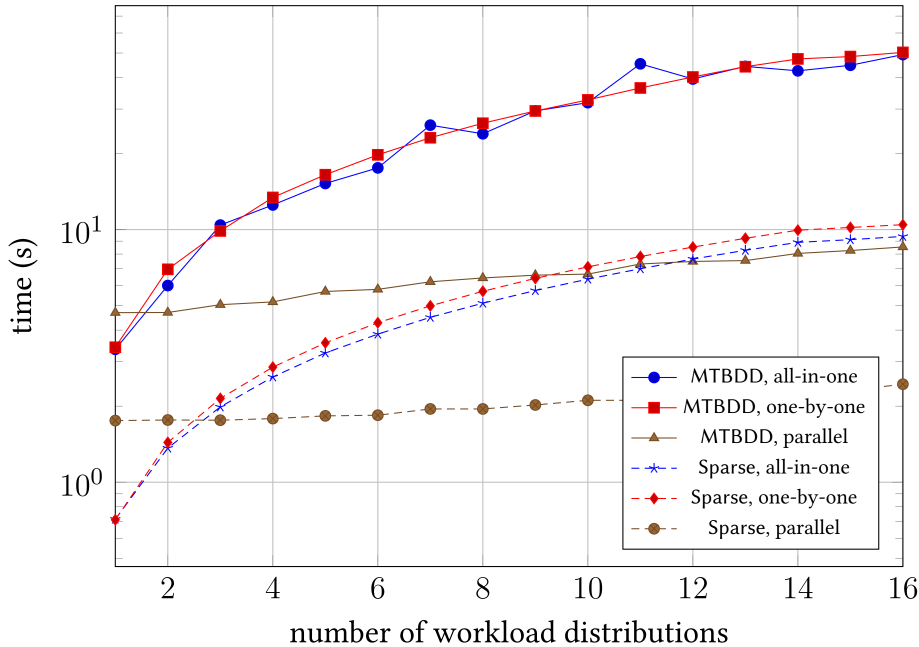


Figure 4.4: Analysis time of the producer-consumer model for an increasing number of different workload distributions

feature module. Overall, there is no clearly superior analysis approach. However, the results indicate that an all-in-one analysis is favorable if the number of family members is high and there is a lot of shared behavior between instances.

4.1.2 Analysis of parametrized systems

The experiments presented in this section serve to investigate the potential of utilizing a family-based analysis for parametrized models. For that, the following models of the PRISM benchmark suite [KNP12] have been converted to ProFeat models: IEEE 802.3 CSMA/CD protocol (Carrier Sense, Multiple Access with Collision Detection), randomized self-stabilizing algorithm, and randomized dining philosophers. In addition, we also consider the Probabilistic-Write/Copy-Select (PWCS) [Bai+13] locking protocol. The PWCS model is defined in terms of two family parameters: The number of writers competing for the access to a shared object, and the number of replicas for a given object.

All models have been originally written in a mix of the PRISM language and some template language. Thus, only a one-by-one analysis was possible. The conversion to ProFeat models required only minor changes to the original models as ProFeat's metaprogramming extensions are very similar to the constructs typically provided by template languages. The source code size of the ProFeat models is comparable to the PRISM template models and in some cases the rewriting into the ProFeat language resulted in a slightly more compact representation.

Table 4.1: Sizes of parametrized models

Model	Instances	MTBDD nodes family	MTBDD nodes separate
CSMA (2–4 processes)	4	633 997	634 076
Self-stabilizing (3–21 processes)	19	4 340	10 662
Philosophers (3–12)	10	82 995	82 689
PWCS (3 replicas, 1–9 writers)	9	134 236	134 190
PWCS (3 writers, 1–7 replicas)	7	955 505	958 033

Table 4.1 presents the model sizes both in terms of family instances and the number of MTBDD nodes that PRISM uses to represent the models. In case of a one-by-one analysis, the number of nodes is the sum of nodes required for representing the separate models for each family instance. A reduction of the number of MTBDD nodes was achieved only for the self-stabilizing model. For all other models, the size of the family model was in the order of the sum of the separate models. This is caused by the fact that there is almost no sharing of behavior between the family members.

Table 4.2 shows the time required for analyzing the models. Each row corresponds to a different query which cover minimal and maximal expected values as well as probabilities for bounded and unbounded reachability. At the time the experiments were conducted, PRISM's Hybrid engine did not yet support the computation of expectations. The results show that the one-by-one approach is almost universally more performant, except for the PWCS model parametrized over the number of replicas. Even for the self-stabilizing algorithm, where a reduction in the number of MTBDD nodes was achieved, the one-by-one analysis is significantly faster. In conclusion, the results indicate that the one-by-one approach is favorable for parametrized models that yield few family instances and that barely have any shared behavior. This matches the experiences made in the previous section where the all-in-one approach was only faster for families with many instances and significant sharing of behavior.

4.2 Software product lines

The concept of features is most prominently applied within the conceptualization, modeling, and implementation of SPLs. In order to demonstrate the feasibility of using ProFeat for the quantitative analysis of SPLs, we consider two "classical" SPL models that have been considered before in the literature. The first case study, a Body Sensor Network product line [Rod+15], shows how ProFeat improves on existing analysis approaches for stochastic family models. Moreover, this case study highlights ProFeat's result post-processing capabilities in a practical setting. Second, the elevator product line [PR01]

Table 4.2: Analysis times (in seconds) for parametrized models (each row represents one query, analysis time for one-by-one is sum over all instances)

	MTBDD			Hybrid			Sparse		
Model	all	1by1	par	all	1by1	par	all	1by1	par
CSMA		timeout			not supported		1 236	1 251	1 220
		timeout		3 660	3 577	3 384	1 078	1 013	954
Self-stabilizing	2 036	1 643	932	251	37	22	129	33	20
	$\ll 1$	1	2		not supported		122	24	15
		timeout			not supported		2 629	476	269
	13	10	7	12	10	6	12	10	7
	13	10	7	13	10	7	13	10	6
Philosophers	9 056	6 212	3 945	9 722	5 949	4 009		out of memory	
PWCS	49	26	15	232	165	130	314	271	220
(over writers)	6 564	2 247	960		not supported		5 473	1 544	1 230
PWCS	752	2 279	1 628	968	348	306	738	2 209	1 265
(over replicas)		timeout			not supported		1 221	3 857	2 735

is extended towards a dynamic feature-oriented system to show ProFeat's potential for analyzing dynamic software product lines.

4.2.1 Body sensor network

A Body Sensor Network (BSN) consists of connected sensors that send measurements to a central entity which evaluates the data to identify conditions critical to the wearer's health. A static BSN product line has been introduced by Rodrigues et al. [Rod+15] where sensors correspond to features. The feature diagram is depicted in Figure 4.5.

The approach of Rodrigues et al. [Rod+15] follows the idea presented in [GS13] to model families of stochastic systems using *parametric* DTMCs [LMT07] to enable a family-based analysis. For each feature, a parameter f is introduced which is 1 if the feature is active and 0 otherwise. A factor p is multiplied to the probability of every transition that depends on the feature, where $p = f$ in case the transition exists if the feature is active and $p = 1 - f$ otherwise. Then, parametric model checking is applied to obtain a single formula which, given a feature combination, evaluates to the probability of reaching a set of goal states. In case of the BSN model, the formula evaluates to the reliability of the system for a given feature combination. The authors of [Rod+15] report that the parametric analysis approach utilizing PARAM [Hah+10] can be seven times faster than a one-by-one analysis using PRISM (for a family size of 298 instances). Furthermore, they proposed a symbolic bounded-search approach that is up to eleven times faster

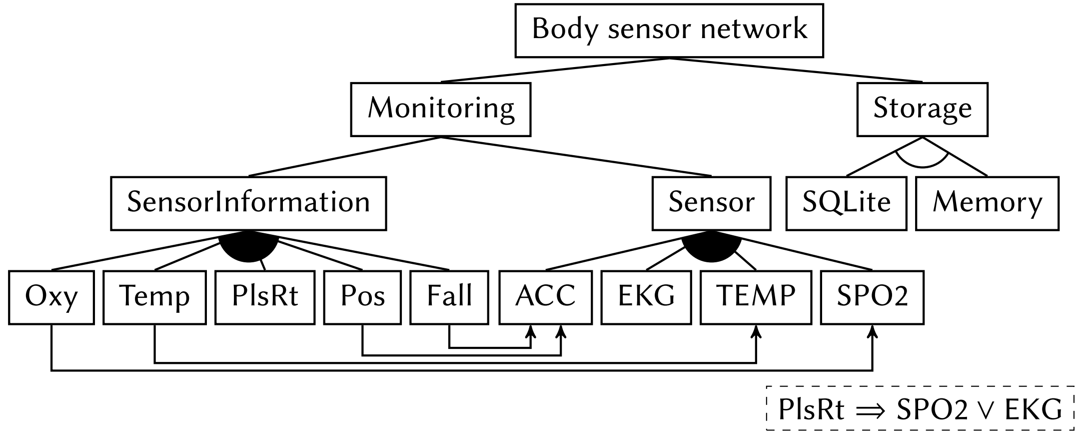


Figure 4.5: Feature diagram for the body sensor network product line

than the one-by-one analysis, and finally demonstrated a handcrafted model-dependent compositional parametric approach that is up to 100 times faster. For obtaining these results, three different model checking tools have been used. Moreover, these existing tools had to be adapted using additional scripts to perform the one-by-one analysis and to evaluate the formulas returned by the parametric model checkers.

With only minor modifications, the parametric BSN model can be turned into a ProFeat model that directly incorporates the feature model of the BSN product line. This is possible since ProFeat's representation of features is directly compatible with the parametric encoding of features as proposed by [GS13]. A feature parameter `f` can be replaced with the expression `active(f) ? 1.0 : 0.0` in the ProFeat model, i.e., if the feature `f` is active, the expression evaluates to 1 and otherwise to 0. Thus, the resulting ProFeat model enables an all-in-one analysis of the same model that has been used by Rodrigues et al. [Rod+15] and furthermore simplifies the comparison to a one-by-one analysis.

With the ProFeat model of the BSN product line, we can reproduce the results of [Rod+15] regarding the reliability of the product line and at the same time compare the performance of an all-in-one analysis using ProFeat with the approaches proposed by Rodrigues et al. The all-in-one approach turned out to be ≈100 times faster than a one-by-one analysis, independent of the used PRISM engine (at 1s for all-in-one vs. 128s for one-by-one with 5651 MTBDD nodes vs. 111507 nodes). Hence, ProFeat directly enables a speed-up of the analysis of the same magnitude as the optimized handcrafted decomposition by [Rod+15].

In the following, the post-processing of ProFeat is applied to gain further insights on the analysis results. For comparison, Listing 4.6 shows an excerpt of the results as produced by PRISM for an all-in-one analysis. After the post-processing by ProFeat, the values of the feature variables are replaced by their corresponding feature names and other system variables are omitted which yields a more readable output as demonstrated in Listing 4.7. Note that ProFeat optionally sorts the feature combinations depending on the analysis results such that the best and worst family instances w.r.t. the query can

be spotted easily. Notably, the ProFeat output looks the same for a one-by-one analysis, whereas a one-by-one analysis using only PRISM produces 298 separate log files with one result each.

```
Results (non-zero only) for filter "init":
2924:(0,1,0,1,0,1,0,1,1,1,1,1,0,0,0,0,1,1,1)=0.9704387384917665
2977:(0,1,0,1,0,1,1,1,0,1,1,1,0,0,0,0,1,1,1)=0.9617396442452253
4041:(0,1,0,1,1,1,1,1,0,1,1,1,0,0,0,0,1,1,1)=0.953118529409139
4191:(0,1,1,0,0,0,0,1,0,0,0,1,0,0,0,0,1,1,1)=0.9792165174854354
5163:(0,1,1,0,0,1,1,1,0,0,1,1,0,0,0,0,1,1,1)=0.9617396442452253
                 ... 293 lines omitted ...

Range of values over initial states: [0.9445746949695,0.9792165174854]
```

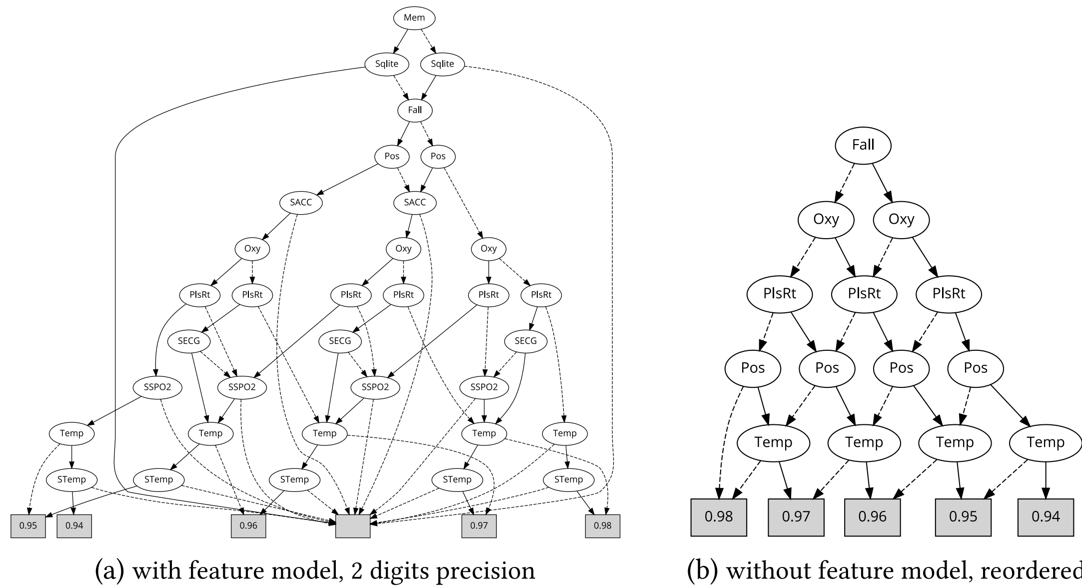
Listing 4.6: Excerpt of the analysis results provided by PRISM for the BSN product-line model

```
Final result: [0.9445746949695,0.9792165174854]
Results for initial configurations:
(Mem, Fall, Oxy, PlsRt, Pos, Temp, SACC, SSPO2, STemp)=0.94457469496953
(Mem, Oxy, PlsRt, Pos, Temp, SACC, SSPO2, STemp)=0.953118529409139
(Mem, PlsRt, Pos, Temp, SACC, SSPO2, STemp)=0.9617396442452253
(Mem, Pos, Temp, SACC, STemp)=0.9704387384917665
(Mem, PlsRt, SECG, STemp)=0.9792165174854354
                 ... 293 lines omitted ...
```

Listing 4.7: Excerpt of the analysis results after post-processing by ProFeat

The symbolic representation of the analysis results (rounded to a precision of two decimals) in the form of an MTBDD is shown in Figure 4.8a. Note that this MTBDD encodes the feature model in addition to the results. The corresponding MTBDD without the feature model is presented in Figure 4.8b which is a much clearer representation of the results than the explicit table with 298 rows. From this representation we can learn that only 5 of the product lines' 11 non-mandatory features actually have a direct influence on the reliability of the system. The other features are either dependent on the inclusion of these 5 features or have no impact on the result at all, like the Mem and Sqlite features. Consider, for example, the leftmost path ¬Fall, ¬Oxy, ¬PlsRt, ¬Pos to the result 0.98. The feature model states that at least one of the sub-features of the SensorInformation feature must be selected. Thus, when following the described path in the diagram, it is clear that the Temp feature must be selected and, in turn, also the TEMP sensor feature. Therefore, the nodes for Temp and TEMP are not present on this path of the diagram. With respect

to the overall reliability of the system, the diagram shows that the system becomes less reliable for each added feature. It is hard to draw such high-level conclusions directly from the explicit result representation. Thus, the transformation of the results into an MTBDD can be a very useful tool for deriving products with the desired properties.



(a) with feature model, 2 digits precision

(b) without feature model, reordered

Figure 4.8: MTBDD representations for the analysis results of the Body Sensor Network product line. Outgoing solid edges denote that the feature is included, dashed edges indicate that it is excluded. The empty terminal node denotes an invalid feature combination.

4.2.2 Elevator product line

The elevator (or lift) product line has been originally introduced by Plath and Ryan [PR01] and subsequently considered as a case study for family-based verification, e.g., in [Ape+13b; Cor+13c]. The system consists of a single cabin which can move to different floors of a building. Persons wishing to take the elevator first have to push a call button on the landing they are on. Once the elevator arrives, a floor button inside the elevator has to be pushed to select the target floor. In its basic version [PR01], the static product line comprises 5 optional features that extend the basic behavior:

Parking. If there are no more landing requests and the elevator is idle, the cabin moves to a designated parking floor instead of remaining in the same position.

$^2/_3$ **full.** If the cabin is filled up to $^2/_3$ of its capacity, it no longer takes any new landing requests until it has served all the passengers inside the cabin.

Overloaded. If the number of persons inside the cabin exceeds the elevator's weight restriction, the doors will not close until enough persons leave the cabin.

Table 4.3: Sizes of elevator models

Model	Instances	MTBDD nodes family	MTBDD nodes separate
Elevator (2 floors)	64	42 254	1 329 204
Elevator (3 floors)	64	151 274	4 924 349
Elevator (4 floors)	64	420 448	13 519 274
Elevator (2–4 floors)	192	779 569	19 772 827

Empty. If the cabin is empty, all requests issued from inside the cabin are cancelled as they are no longer relevant.

Executive floor. This gives the requests from a designated floor priority over other requests.

For the case study considered in this section, several extensions have been added to the model. First, nondeterministic choices are resolved with probabilities where appropriate, e.g., to model the request rate and to add a probability of failure. Second, an additional service feature enables the system to automatically call service technicians that may repair the elevator or install new features. As a consequence, the elevator system we consider here is actually a dynamic product line where features can be activated and deactivated during runtime. We deal with a simple instance of the elevator system which can transport one person and where at most two persons are in the system. Additionally, the system is parametrized by the number of floors (2–4). We also consider the family of three product lines, comprising 192 instances of the elevator system (64 valid feature combinations and 3 possible parameter values). The sizes of these models are shown in Table 4.3. The much smaller number of MTBDD nodes required for representing the family model indicates that a lot of behavior is shared between the family instances. In the quantitative analysis of the model, we asked whether the minimal (i.e., worst case) probability to successfully serve the top floor within the next three steps if the cabin is initially at the first floor and the top floor is requested is greater than 0.99. The performance of the analysis approaches and PRISM engines is shown in Table 4.4. Especially for larger instances, the all-in-one analysis using the MTBDD engine considerably outperforms all other approaches and engines. Thus, we again reach the conclusion that the all-in-one approach is particularly effective for product-line models with many instances.

4.3 Self-adaptive systems

This section presents two case studies that serve to illustrate how the language constructs provided by ProFeat are useful beyond modeling classical product lines and parametrized

Table 4.4: Analysis times (in seconds) for elevator models (analysis time for one-by-one is sum over all instances)

	MTBDD			Hybrid			Sparse		
Model	all	1by1	par	all	1by1	par	all	1by1	par
Elevator (2 floors)	1	65	7	2	49	7	1	45	7
Elevator (3 floors)	4	223	11	98	2 531	96	7	286	18
Elevator (4 floors)	15	910	32	2 601	54 262	1 952	56	2 008	83
Elevator (2–4 floors)	29	1 199	49	5 089	56 843	2 052	74	2 339	106

models. In particular, a feature-oriented network system model shows how using ProFeat is beneficial for analyzing single systems where the concept of features is applied to describe the dynamics of (self-)adaptivity. In the second case study, the verification of an adaptation protocol for self-adaptive systems, the combination of feature-oriented modeling and ProFeat's metaprogramming capabilities proved to be effective for generically describing the system.

4.3.1 Adaptive network system model

This case study illustrates how the concept of dynamic features can be applied to simplify the modeling of an adaptive system. We consider a distributed system which consists of several processing elements (PEs) connected by a heterogeneous network for communication. The network comprises links with different characteristics, as shown in Figure 4.9, where PEs are connected by wired links and directed wireless links. While the wired links are static and always available, the wireless links can be dynamically switched on or off. Furthermore, these link types differ in their throughput and energy consumption. We assume that both activating wireless links and keeping them active consumes energy, even in case they are unused, i.e., a turned-on link carrying no network traffic is still considered to be active. In the model, the system runs concurrent tasks that may be distributed among several PEs and the subtasks processed by the PEs utilize the network links to communicate. We focus on the network part of the system and thus abstract from the operational behavior of the PEs. Therefore, the state of the network system is solely determined by the load (network traffic) on the communication links.

The operation of the model is structured into four phases. First, a new task is generated. The number of PEs necessary to process the task is given by a binomial distribution which allows us to determine the load on the system using a single parameter. In the second phase, the task is mapped onto the PEs which subsequently increases the load on the network links connecting the selected PEs. The mapping may cause wireless links to be activated if necessary. If no mapping is found or no mapping is possible at all, the task is dropped, raising a "fail" event. Within the third phase, wireless links may be activated

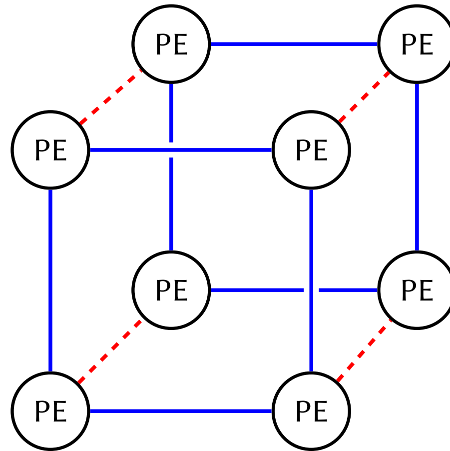


Figure 4.9: A network system with cube topology. The solid lines denote wired connections, while the dashed lines indicate directed wireless connections.

and, given that they carry no network traffic, may also be deactivated. The processing of tasks happens in the fourth phase. Finishing tasks is modeled by probabilistically decreasing the load on the network links. Once the last phase ends, the described process repeats from the start.

There are several possibilities to implement the task mapping in such a system. In this case study, three variants of the system were considered. In the first two, the task mapping is nondeterministic, enabling a best- and worst-case analysis. The third variant utilizes a randomized strategy that may serve as a basis for an actual implementation of the task mapping.

Nondeterministic. For a task that requires n network links (and $n + 1$ PEs), this model nondeterministically selects one of all possible spanning trees with n edges over the network topology.

Nondeterministic with hopping. This is a variation of the nondeterministic model where mappings may include additional *hops*, i.e., PEs that do not contribute to the processing of a task and merely serve as communication relays. For each generated task, number of hops is chosen according to some fixed probability distribution.

Heuristic. A simple heuristic for mapping tasks applies the "greedy" principle. First, the PE with the most capacity remaining among its connected network links is selected. Then, the task is randomly distributed over the adjacent PEs. The probability that an adjacent PE is selected is indirectly proportional to the load on the adjacent link.

In the following, we discuss some notable modeling details. Each wireless link i is treated as an optional dynamic feature `Link[i]`, i.e., the set of wireless links is compactly represented by a multi-feature. As a consequence, the definition of costs is simplified,

as they have to be defined only once in the feature model and are then automatically applied to all links. Turning wireless links on and off is handled by the feature controller which activates and deactivates the corresponding features. An excerpt of the controller in the nondeterministic variant is shown in Listing 4.10. A `for` loop is used to generate a deactivation command for each wireless link. The `load` array stores the current amount of network traffic for each link. The excerpt in Listing 4.11 shows a part of how the mapping is realized in the model. Here, one of all possible spanning trees of size `task_size` is selected. An element with index i in the `selected` array is `true` if the PE with index i is a node in the spanning tree. The parametrized formulas `from(k)` and `to(k)` return the endpoints of a link with index k. The spanning tree is built incrementally where in each step another link, which must be connected to any PE that has been chosen beforehand, is selected nondeterministically. The `load` of the selected link is increased accordingly and the remaining task size to be mapped is decremented. The process repeats until the task size reaches zero. The metaprogramming constructs allow that the behavior of the model is defined completely independent of the network topology. Instead of being hard-coded into the model, the topology is specified by a constant array which can be accessed using the `from` and `to` formulas.

```
1 controller
2   // ...
3   for i in [0..NUM_OPTIONAL_LINKS-1]
4     [] active(Link[i]) & load[i] = 0 & phase = RECONF & link = i ->
5         deactivate(Link[i]);
6     [] phase = PHASE_RECONF & link = i -> true;
7   endfor
8 endcontroller
```

Listing 4.10: Part of the network system's feature controller that nondeterministically deactivates unused wireless links

```
1 for i in [0..NUM_LINKS-1]
2   [] phase = MAP & task_size > 0 & load[i] < link_capacity(i) &
3      selected[from(i)] & !selected[to(i)] ->
4      (selected[to(i)]' = true) & (load[i]' = load[i] + 1) &
5      (task_size' = task_size - 1);
6 endfor
```

Listing 4.11: Excerpt of the nondeterministic variant of the network system model showing how a spanning tree is built iteratively during the mapping phase

In the model that has been used for the subsequent quantitative analysis, the maximal task size was limited to 2. Here, the task size refers to the minimal number of network links the task requires, i.e., a task of size n needs at least n links and $n + 1$ PEs. The probability that the next incoming task has size 1 is given by the parameter t_1. Thus, the probability t_2 for an incoming task of size 2 is $t_2 = 1 - t_1$. The minimal probability that mapping a task fails within the first 9 rounds depending on t_1 are presented in Figure 4.12. The results for the nondeterministic variant represent the theoretical optimum. As expected, the probability of a mapping failure decreases if the arrival of a task with size 2 becomes less likely. Tasks of size 1 can be mapped easily if there is at least one link that still has enough spare capacity. Furthermore, the ability to map a task of size 1 does not depend on previous mapping choices. However, mapping of a task with size 2 is more likely to fail, as two links connected via a PE have to be found that still can accommodate the additional load. Thus, whether mapping a task of size 2 can be successful also depends on previous mapping choices.

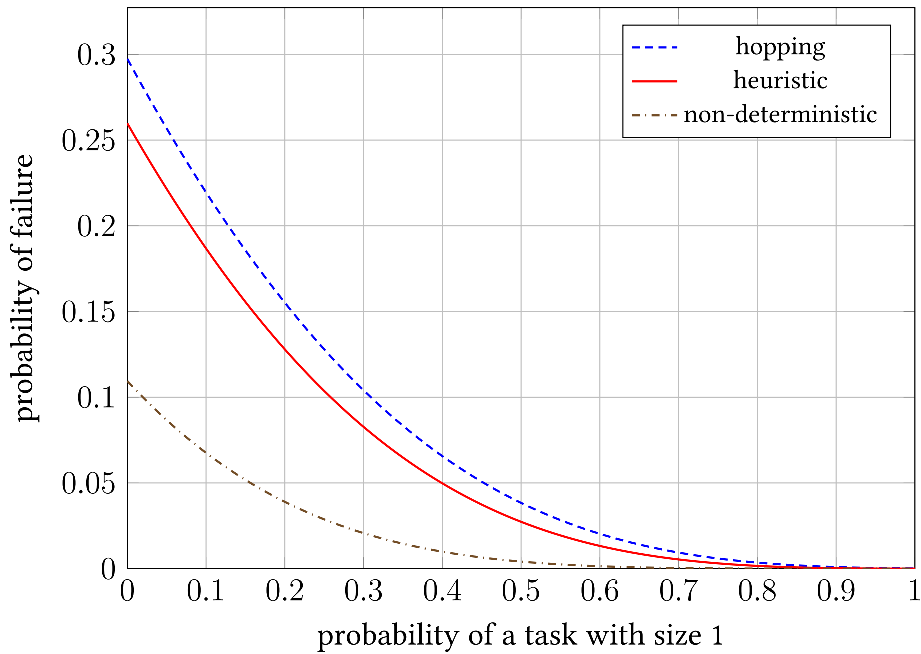


Figure 4.12: Minimal probability of that a mapping failure occurs within 9 rounds, depending on the arrival probability of a task with size 1 (instead of a task with size 2) in each round

The second experiment considers the trade-off between successful mappings and the energy required to guarantee them within a certain probability which is formalized as an energy-utility quantile [Bai+14]. Here, we assume that the activation of a wireless link consumes 2 units of energy and keeping them active requires 1 unit. Wired links do not incur any additional energy costs. Figure 4.13 shows the quantile value as the minimal energy required depending on the probability that at most one mapping fails. The higher the probability, the more likely it is that wireless links must be utilized to avoid mapping

failures and hence more energy must be used. The plot shows the results for $t_1 = 0.5$, i.e., the distribution of task sizes is uniform. Due to the combinatorial blow up of possible mappings, the model suffers from the state-space-explosion problem with around 120 million states for the topology shown in Figure 4.9. Therefore, the analysis had to be carried out using PRISM's MTBDD engine.

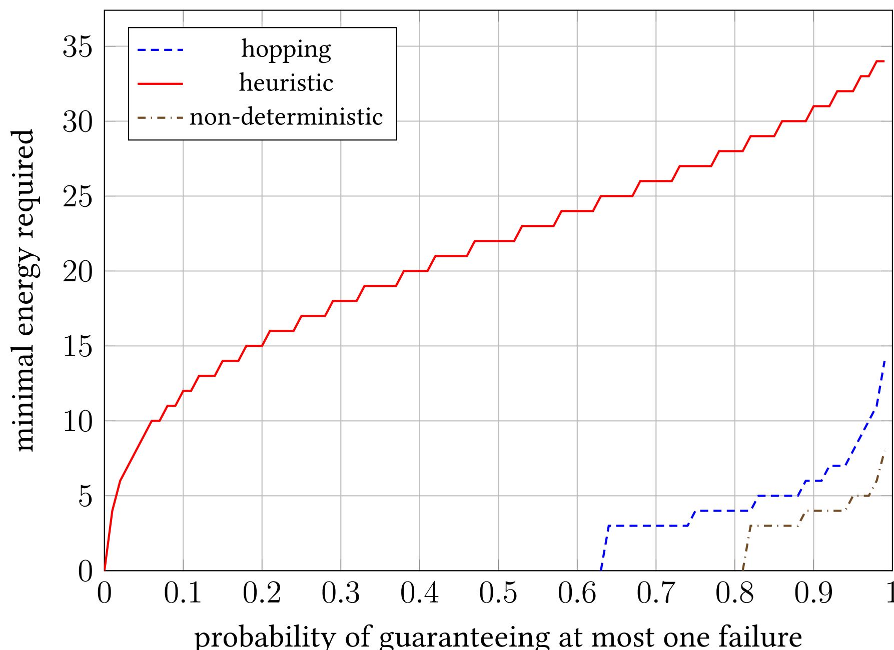


Figure 4.13: Minimal energy required such that there is a mapping strategy which guarantees with probability p at most one mapping failure within 9 rounds

4.3.2 Adaptation protocol for distributed systems

The focus of this case study was the verification and quantitative analysis of a novel protocol for coordinating adaptations in a distributed system [Wei+17]. Such a system consists of several collaborating nodes that can communicate over a possibly unreliable network. We assume that on each node the application logic is strictly separated from the adaptation logic which is encapsulated in the local *adaptation manager*. Due to changes in the context or environment, new goals, or occurring defects, the distributed system may need to adapt in order to react to changed circumstances. We further assume that there is no central entity coordinating the adaptation. Rather, all nodes are required to coordinate their local adaptations with the other involved nodes such that eventually a consistent system state is reached. That is, either all nodes have successfully applied the adaptation, or, in case of some unexpected error, the previous system state is restored. A decentralized approach for adaptations complicates the coordination of the individual

nodes, but it makes the system more resilient by not relying on a centralized coordination entity.

The coordination protocol for decentralized adaptation of multiple distributed nodes proposed in [Wei+17] proceeds as follows. First, the adaptation plan (generated by some planning component) is sent to all involved nodes and provides them with two important pieces of information: the set of other nodes that are involved in the adaptation, and which local adaptations are necessary on each node. In a second step, the involved nodes must reach a stable state that allows for the local adaptations to be applied. After that, each node locally applies the adaptations. If a node was able to successfully change its configuration, it sends a *report* message to every other involved node and enters a waiting state. It remains in this state until it has received the *report* messages of all other involved nodes. If this finally happens, the whole adaptation is considered to be successful. In case a node could not apply its local adaptation because of some error, it sends a *transaction rollback* message to the other nodes to abort the adaptation across the system. Obviously, the coordination messages of the protocol can get lost in an unreliable network. If some node A did not receive the expected *report* message of some node B until a timeout has occurred, node A will first request that node B resends its *report* message. In case that is unsuccessful, node A will send a broadcast message to all other nodes asking for a confirmation of B's success. If that also fails, A will abort the adaptation locally and send the *transaction rollback* message to all other nodes. The protocol further allows grouping multiple adaptations together into an *adaptation transaction* to ensure a consistent system state.

Modeling details

The support for feature-oriented modeling combined with the metaprogramming facilities of ProFeat enables a generic definition of the system's and the protocol's operational behavior. For each node i of the system, the model contains a feature `Node[i]`, i.e., the set of nodes is represented as a multi-feature. Thus, the node behavior needs to be defined only once and is automatically instantiated for all nodes. For realizing asynchronous message passing between the nodes, the model contains a `Network` feature that stores all incoming messages in a buffer, as shown in Listing 4.14. ProFeat allows action labels to be indexed like arrays (line 6) which enables a general definition of the network feature module independent of the concrete messages that are being sent. If a message arrives, it is placed into the next empty cell of the buffer (lines 6–8), or it is lost with a certain probability (line 9). Conversely, if a node receives a message from the network, the message is removed from the buffer (line 11). Note that the network imposes no order on sending and receiving, thus messages may also get reordered. The index of the `send` and `recv` messages carries the actual message content and is generated using parametrized formulas for each message type. Consider for example Listing 4.15 where a node sends a report message from itself to another node (with sender and receiver given as arguments

```
1 module Network(buf_size) {
2   cell : array [0 .. buf_size - 1] of [EMPTY .. NUM_MSGS - 1] init EMPTY;
3
4   for msg in [0 .. NUM_MSGS - 1]
5     for c in [0 .. buf_size - 1]
6       [send[msg]] for i in [0 .. c - 1] cell[i] != EMPTY & ... endfor &
7         cell[c] = EMPTY ->
8           (1 - P_MSG_LOSS): (cell[c]' = msg) +
9           P_MSG_LOSS: true;
10
11      [recv[msg]] cell[c] = msg_id -> (cell[c]' = EMPTY);
12    endfor
13  endfor
14 endmodule
```

Listing 4.14: The `Network` module uses a buffer for realizing asynchronous message passing

to the `report` formula). The model is parametrized by number of nodes in the network, the buffer size, and the probability for message loss. Furthermore, a `Scenario` feature describes the adaptation transaction and can be configured by a set of constants. Overall, the generality of the model allows us to quickly experiment with different settings and scenarios without having to rewrite the model every time.

```
1 for i in [0 .. NUM_NODES - 1]
2   [send[report(id, i)]] state = S_SEND & index = i -> (index' = index+1);
3 endfor
```

Listing 4.15: Excerpt of the `Node` feature module for sending a report message over the network

Quantitative Analysis

After having verified that the protocol is deadlock-free and that adaptations are applied consistently in case there is no message loss, we asked for the minimal probability of a successful adaptation transaction for a probability of message loss between 0 and 0.1. The results for a system with 3 nodes and 2 adaptation steps are presented in Figure 4.16. The analysis has been carried out for a model variant with no handling of message loss and another variant where a node will actively request the resending of lost *report* messages. As expected, the latter protocol variant is more resilient to message loss.

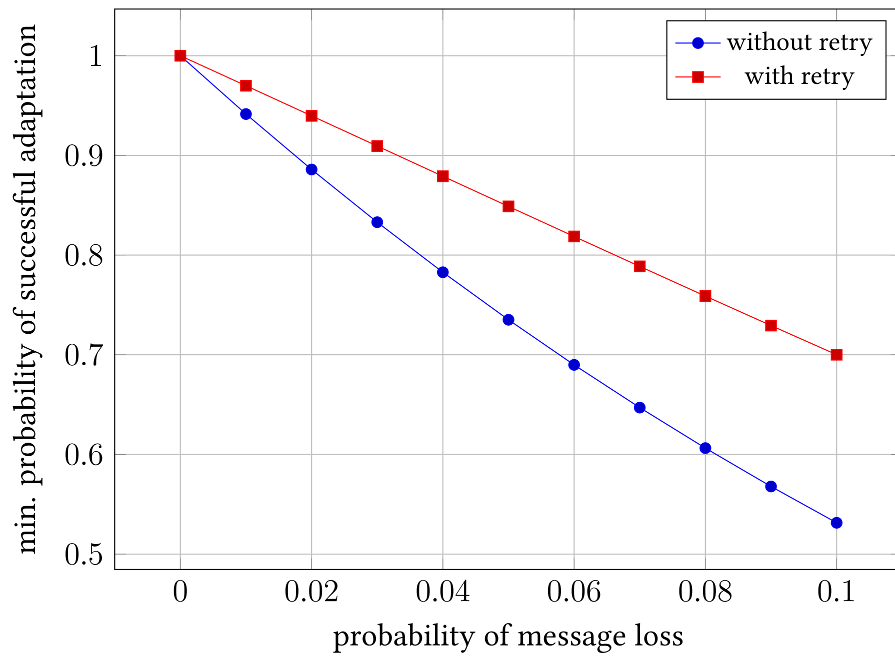


Figure 4.16: Minimal probability of a successful adaptation transaction for a given probability of message loss

Part II

Role-based systems

5 Formal modeling and analysis of role-based systems

In this chapter, a compositional modeling framework for context-sensitive and adaptive systems is proposed. For that, we adopt concepts and notions from role-based modeling, as roles elegantly capture context-specific adaptations. The modeling formalism comprises *role-based automata* (RBA) which provide roles as a first-class modeling concept. Role-playing, i.e., the active enactment of role-related behavior, is made explicit by role-playing annotations which enable reasoning about the active participation of components within (unintended) interactions. This is especially important in highly dynamic, adaptive systems where the occurrence of interactions may depend on the sequence of role activations and deactivations. Furthermore, roles make the context in which components act explicit as well. This not only allows us to reason about interactions within single systems, but also about interactions between systems or within hierarchies of systems.

Outline. The first part of this chapter, Section 5.1, gives an overview of the role concept with a focus on conceptual modeling and role-based programming languages. Based on the concepts and notions of role-based modeling, the second part (Section 5.2) introduces RBA which allow capturing role-based behavior, such as dynamic role-playing and role interactions, within formal operational models.

The contents of Section 5.2 are based on the publication [Chr+20].

5.1 The role concept

The role concept has been considered in a broad range of fields [ZZ08], including data modeling, conceptual modeling [Küh+14; Ste00], programming languages, security (role-based access control [FK92]), multi-agent systems [Cab+10], and computer-supported collaborative work. Bachman and Daya introduced roles in data modeling as early as 1977 to capture evolving entities within databases [BD77]. Distinguishing entities and roles that entities might play enables a more natural mapping of real-world concepts to the data model and allows a more flexible modification of data. Consider the example of a person (the entity) who is an employee (the role) in a company. The person's record may contain the full name, date of birth, etc., while the employee record stores the position, salary, office telephone number, and so on. Separating these records is justified by the

fact that they have different lifetimes. An employee might leave a company or may change the department which requires the employee record to be deleted or updated. A person, however, does not cease to be a person if the employment changes. Furthermore, a person may have multiple part-time employments which is easily handled by adding another employee role. Naturally, a person entity can acquire different roles depending on the context. For instance, the person might additionally become a project manager. By treating roles as a first-class concept, transient and context-dependent properties can be handled without modification of entity records, facilitating an evolution of data.

By encapsulating context-dependent behavior and properties, roles facilitate a separation of concerns. A collaboration between objects can be modeled using a set of interacting roles where each role encompasses the behavior of a participant in the collaboration. With roles acting as intermediaries between objects, relationships become explicit, making the complex interactions between objects easier to grasp. This is in contrast to standard object-oriented systems where relationships are only implicitly defined, usually by references. However, roles are not only conceptually relevant, but also provide a practical programming mechanism. Since they encapsulate behavior and are able to adapt objects, they enable a fine-grained modularization. Not only does this make the implementation of a system more manageable, it also facilitates component reuse. For instance, consider a leader election protocol in a distributed system. This collaboration, represented by a set of roles, may be reused across different (unrelated) components of the system or even in the implementation of other systems. It has been observed that roles share similarities with aspects from aspect-oriented programming [Kic+97]. In particular, both concepts allow the modification of existing behavior. That makes roles suitable for implementing cross-cutting concerns [GØ02; Her02; HU02; VL16]. A concern is called cross-cutting if its implementation is scattered over multiple components and thus cannot be cleanly modularized using standard object-oriented techniques. Examples include monitoring, synchronization, and transaction management. In conclusion, roles are not only conceptually appealing, but serve a practical purpose by enabling a separation of concerns. With that, systems can be built compositionally by combining small, manageable parts.

5.1.1 Towards a common notion of roles

Roles are an intuitively understood concept and have been widely recognized as a useful modeling construct, as they allow for an accurate representation of real-world concepts. However, most approaches that incorporate roles introduced a different definition of roles, often without considering already existing work. This is not overly surprising, as each research field had different reasons for introducing roles, fulfilling their specific needs and requirements. Overall, this lead to a fragmented research landscape and still to this day there is no consensus on what a role actually *is*. Roles have been considered to be named places in relationships, a collection of dynamic properties of objects, a description

of context-dependent attributes and behavior, and have been characterized as a set of capabilities, requirements and goals. As a result, there have been several attempts to reconcile and unify the different role notions that have been presented in the literature, most prominently by Steimann. In a survey of role-based approaches [Ste00] that have been proposed before the year 2000, he identified the following list of 15 characteristics associated with roles.

1. A role comes with its own properties and behavior.
2. Roles depend on relationships.
3. An object may play different roles simultaneously.
4. An object may play the same role several times, simultaneously.
5. An object may acquire and abandon roles dynamically.
6. The sequence in which roles may be acquired and relinquished can be subject to restrictions.
7. Objects of unrelated types can play the same role.
8. Roles can play roles.
9. A role can be transferred from one object to another.
10. The state of an object can be role-specific.
11. Features of an object can be role-specific.
12. Roles restrict access.
13. Different roles may share structure and behavior.
14. An object and its roles share identity.
15. An object and its roles have different identities.

Note that the term "feature" in item (11) is not restricted to the notion of features in the feature-oriented approach, but refers to any operation or functionality provided by a role-playing object. Also note that some of these characteristics are conflicting, e.g, (14) and (15), thus no approach can provide all characteristics at once. Based on his observations, Steimann proposed the general role-based modeling language *Lodwick* [Ste00] to serve as a foundation for other role-based modeling languages and programming languages.

A more recent survey of role-related approaches has been conducted by Kühn et al. [Küh+14]. With the rise of mobile and interconnected devices, the focus of contemporary approaches has shifted to address context-aware systems and applications. Consequently, several proposed modeling languages and programming languages include *context* as a first-class modeling construct. Since the term "context" is both vague and heavily overloaded, these approaches often introduce a new notion for this concept, such as *team* [Her02], *institution* [Gen07], or *ensemble* [HK14]. Kühn et al. proposed the general term *compartment* [Küh+14] in order to unify these different notions of objectified contexts. To account for the context-related characteristics of roles, they extended the list of Steimann with the following items.

16. Relationships between roles can be constrained.
17. There may be constraints between relationships.

18. Roles can be grouped and constrained together.
19. Roles depend on compartments.
20. Compartments have properties and behavior.
21. A role (type) can be part of several compartments.
22. Compartments may play roles like objects.
23. Compartments may play roles which are part of themselves.
24. Compartments can contain other compartments.
25. Different compartments may share structure and behavior.
26. Compartments have their own identity.
27. The number of roles occurring in a compartment can be constrained.

In addition to the extended list of role characteristics, Kühn et al. classified the approaches in the literature by which of the three different *aspects* of roles[1] they address. The *behavioral* aspect refers to the capability of roles to extend or alter the structure and behavior of an object. Modeling languages which support relationships as a first-class citizen emphasize the *relational* aspect of roles. Here, roles depend on and are defined by the relationships between them. *Context-dependent* roles encapsulate attributes and behavior that are relevant in a specific collaboration, situation, or protocol. From these three aspects and their combinations, the following classes of role-based approaches arise.

Behavioral aspect. The *Generic Role Model* [DPZ02] addresses behavioral and structural adaptation only. An emphasis on behavioral adaptation can also be found in many role-based programming languages such as *Chameleon* [GØ03], *Rava* [He+06], and *JavaStage* [BA12].

Relational aspect. In *UML* class diagrams [OMG11] and the *Object-Role Modeling (ORM)* approach [Hal05], roles designate the ends of relationships between objects. The programming language *Rumer* [BG10; BGE07] incorporates relationships as a first-class language construct.

Relational and behavioral aspects. Examples of this class include the modeling language *Lodwick* [Ste00] and *OntoUML* [Gui+04; GW12], an extension of the UML based on a precise ontological foundation.

Context-dependent and behavioral aspects. Representatives of this class are the *Metamodel for Roles* [Gen07] which attempts to reconcile earlier role-based approaches, and the *Data, Context and Interaction (DCI)* paradigm [CR14]. Within DCI, all behavior is contained in roles that reside within contexts. The programming language *EpsilonJ* [TUI05] provides first-class contexts and has been extended to *NextEJ* [KT10] which supports scoped context activation as introduced by the

[1] Kühn et al. use the term *natures of roles*.

context-oriented programming paradigm. The *powerJava* language [BBT06] utilizes roles for the exogenous coordination of objects. Arguably the most mature role-based language is the Java extension *ObjectTeams/Java* [Her02; Her07].

Context-dependent and relational aspects. The *Information Networking Model* [LH09] is a data modeling language that extends the classical Entity-Relationship model with roles and contexts. In the *Helena* approach [HK14], components play roles in ensembles. Relationships are made explicit by role connectors. While roles have behavior, components serve merely as providers for data and operations to be invoked by roles. Therefore, structural and behavioral adaptation is not supported.

Combination of all aspects. The *Compartment Role Object Model* (CROM) [Küh+14] is a family of meta-models that allows the combination of all three aspects of roles. The Scala library *SCROLL* [LA15] facilitates role-based programming and, as it is based on the CROM, covers all aspects of roles.

As the CROM currently is the most comprehensive meta-model that combines and unifies many ideas of previous works, the formal modeling approach presented in this thesis takes inspiration from the CROM and its concepts. Therefore, the next section discusses the meta-model in more detail.

5.1.2 The Compartment Role Object Model

The CROM is a meta-model for the conceptual modeling of role-based systems and combines the behavioral, relational and context-dependent aspects of roles [Küh+15]. In fact, it does not only provide a singular meta-model, but rather a family of meta-models. For that, the characteristics of roles listed in Section 5.1.1 are treated as features. To capture dependencies, e.g., that constraining relationships requires that relationships are supported, and other constraints between these features, a feature model in the form of a feature diagram is utilized to define the valid feature combinations [Küh+14; Küh17]. By selecting a specific combination of features, users of the meta-model can customize it and tailor it to their specific needs. Tool support for configuring the meta-model as well as conceptual modeling is provided by the model editor FRaMED [Küh+16].

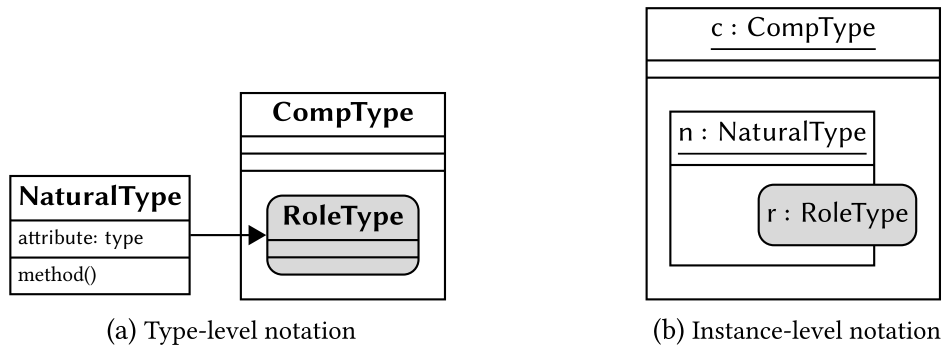
The CROM provides four meta-types: natural type, role type, compartment type, and relationship type. Naturals (e.g., person, company) are entities that are independent of any context and constitute the role players. The context in which roles are played is provided by a compartment (e.g., university, business transaction). Relationship types are used to relate two roles that are dependent on each other and cannot exist in isolation (e.g., a buyer role and a seller role). In order to give these types a precise ontological semantics and to provide guidance on which meta-type should be used for a certain real-world concept, the meta-types are characterized in terms of the meta-properties identity, rigidity, and foundedness. The *identity* [Gui05; GW00] property states whether

the instance of a type has a unique, derived, or composite identity. For instance, a person has a unique identity that is not shared with or derived from any other entity. The identity of a student role, on the other hand, is derived from the identity of the person playing it. Relations between entities often have a composite identity, e.g., a business transaction might derive its identity from the buyer, seller, and the items being sold. If an instance has a *rigid* [GW00] type, then it has this type during its whole lifetime. For instance, a car cannot cease to be a car until it is scrapped. The dual, anti-rigidity, denotes that a property can cease to hold, e.g., a person can stop being a customer without ceasing to be a person. Finally, *foundedness* [GW00; MKK12] (also called *dependence*) describes entities whose existence depends on another entity, e.g., to be a customer there must exist another person who is a vendor. Furthermore, both the customer and the vendor depend on the context, i.e., the exchange of money and goods. Using these three ontological properties, each concept provided by the CROM is characterized. For instance, a role type derives its identity from its player. Furthermore, it is anti-rigid, as a role can cease to exist without affecting the lifetime of its player. Finally, roles are founded, as they depend both on the existence of a player and a context. The characterization of all meta-types is summarized in Table 5.1.

Table 5.1: Ontological characterization of meta-types [Küh+15]

Concept	Identity	Rigidity	Foundedness	Example
natural type	unique	rigid	non-founded	person, car
role type	derived	anti-rigid	founded	student, customer
compartment type	unique	rigid	founded	university, transaction
relationship type	composite	rigid	founded	advises, married

The CROM framework provides a graphical notation for models, i.e., the type level, as well as for instances [Küh+15]. We will only discuss the subset of notations here that are used throughout this thesis. For a full reference of the graphical notation, see [Küh17]. The type-level notation, which is inspired by UML class diagrams, is shown in Figure 5.1a. Natural types and compartment types are represented by boxes where the first line constitutes the type name. Optionally, attributes and method signatures can be defined. Role types are defined similarly, but are represented by rounded boxes to visually distinguish them from rigid types. The “fills” relation ($\longrightarrow$) specifies that a type can fill a certain role type, i.e., its instances can act as instances of the indicated role type. Compartment types have an additional part that contains the participating role types. The instance-level notation is shown in Figure 5.1b. In a *Compartment Role Object Instance* (CROI), instances are drawn analogously to their corresponding types. For a visual distinction from types, the instance name is underlined and followed by the type of the instance. Again, roles are drawn with rounded corners. That a role is played by an instance is denoted by drawing the role’s box inside the role segment of the player’s box,



(a) Type-level notation

(b) Instance-level notation

Figure 5.1: CROM graphical notation (extracted from [Küh17])

e.g., the instance n plays the role r. Compartment participation is likewise indicated by putting the role (and also its player) into the lower part of the compartment.

To summarize, the CROM combines the behavioral, relational, and context-dependent aspects of roles and provides a clear graphical notation for the conceptual modeling of role-based systems. The formal framework presented in the latter part of this chapter therefore adopts the ontologically founded notions of naturals, roles, and compartments as defined by the CROM.

5.1.3 Roles in programming languages

For a modeling formalism aiming to capture the operational behavior of role-based systems, the behavioral aspect of roles is especially important. Therefore, the focus of this section is to survey existing role-based programming languages to gain insights on which effects roles can have on the behavior of objects, components, and systems.

We can distinguish three groups of role-based programming languages w.r.t. the possible adaptations a role can apply to the behavior of its player: no adaptations are possible, only new behavior can be added, and lastly, arbitrary modifications of behavior are possible.

No adaptation. Within *JavaStage* [BA12], roles are mainly utilized for modularization and the separation of concerns. Hence, only static roles are supported and no adaptation mechanism is provided. In *powerJava* [BBT06], roles are a means for the exogenous coordination of components. As such, they constitute a two-way interface between the component and the context: methods that must be provided by the component and methods that are provided to the component by the environment. Since roles have no own behavior in this language, no adaptation is possible.

Add new behavior. The only language falling into this group is *Rumer* [BG10; BGE07]. Here, roles are defined as parts of relationships. A relationship may add properties and new behavior to the components it connects, but cannot alter their behavior.

Modify behavior. Role-based programming languages that focus on the behavioral aspect of roles and use them as a construct for capturing context-dependent behavior belong to this group. *Chameleon* [GØ03] applies a well-known adaptation mechanism from aspect-oriented programming [Kic+97]. A role can inject behavior before and after existing methods of the player. Methods can also be overridden, i.e., replaced completely, by the role. In case both the player and one or more of its roles provide a method with the same name, the method with the highest priority is chosen upon invocation. These priorities can be changed dynamically to adapt to different contexts. *ObjectTeams/Java* [Her02; Her07] applies aspect-oriented techniques as well. *Rava* [He+06] internally uses the role-object pattern [Bäu+98] and thus also allows replacing methods of the player by role methods. Both *EpsilonJ* [TUI05] and *NextEJ* [KT10] likewise utilize a technique reminiscent of the role-object pattern. The *SCROLL* library [LA15] uses a yet another approach. Here, the player and its roles are internally stored as nodes in a graph. A dispatch configuration specifies how this graph is traversed to find the target of a method invocation. Clearly, by prioritizing the role method over the player's method, replacement of the player's original behavior is possible. Note that in all languages in this group it is possible to invoke the base behavior of the player from within the role's behavior, e.g., to extend and adapt already existing functionality.

In conclusion, most role-based programming languages focusing on context-dependent behavior support a flexible adaptation by replacing existing behavior and adding new behavior. Consequently, the formal approach for modeling the operational behavior of role-based systems presented in the following allows adding and overriding behavior as well.

5.2 Compositional modeling of role-based behavior

In this section, we introduce an automata-based formalism for modeling the operational behavior of role-based systems. The conceptual notions used here are inspired by the CROM [Küh+15] which unifies the well-established views on roles presented in the literature. Similar to the CROM, we distinguish between different kinds of components, namely *naturals* that constitute the role players, *roles*, and *compartments* that are objectified contexts in which the roles are played. In contrast to the CROM and other similar meta-models which are only concerned with the structure of a role-based system, we focus here on their operational behavior. For that reason, we do not consider role relationships, as they are mainly a means to describe and constrain the structure of role-based systems.

As the separation of concerns is a major motivation of the role-oriented development approach, role-based systems naturally admit a compositional structure with naturals, roles, and compartments as the basic components. We introduce RBA as a uniform representation of such components. These basic components are combined using operators for

role-binding and parallel composition. *Role-binding* enables a player to act in a certain role, i.e., to *play* the role. The distinction between role-binding and role-playing, corresponding to the *ability to act* in a certain role and actually *playing* the role, respectively, has also been considered by Mizoguchi et al. [MKK12]. To attain a joint automaton capturing both player behavior and emerged role-playing behavior, the role-binding operator takes an RBA representing either a natural, a role, or a compartment, and combines it with an RBA specifying the role-dependent behavior. The possibility to bind roles to compartments, which themselves consist of role-playing components, enables a hierarchical modeling of systems. The second operator, *parallel composition*, formalizes the interaction of RBA using the standard notion of synchronization over shared actions. Applying the aforementioned composition operators for combining the RBA of all components results in a single RBA encompassing the behavior of the whole role-based system. This RBA also contains all possible combinations of role-playing. However, usually not all role-playings are actually allowed. Constraints regarding role-playing are formalized using another automata-based component, the *(role-playing) coordinator*. The composition of an RBA with a coordinator resolves all allowed combinations of role-playing and yields an MDP. As MDPs are a well-established formalism with broad applications, this semantics enables the application of already available tool support.

The concepts presented in this section are illustrated using a banking example adapted from [CR14] and [Küh+15]. In this scenario, an Account (a natural type) implements only the basic functionality of a bank account: storing a balance and providing operations to increase or decrease it. The (context-specific) functionality for a money transfer between accounts is encapsulated in the Source and Target roles within the Transaction compartment, resulting in a separation of concerns. Similarly, the business rules for different kinds of accounts is captured in roles within the Bank compartment. An account playing the Checking account role may be overdrawn up to a certain limit. A fee is applied to every transaction on a Savings account. The role model of the example is shown in Figure 5.2, depicting that a natural of type Account can fill the roles of type Checking,

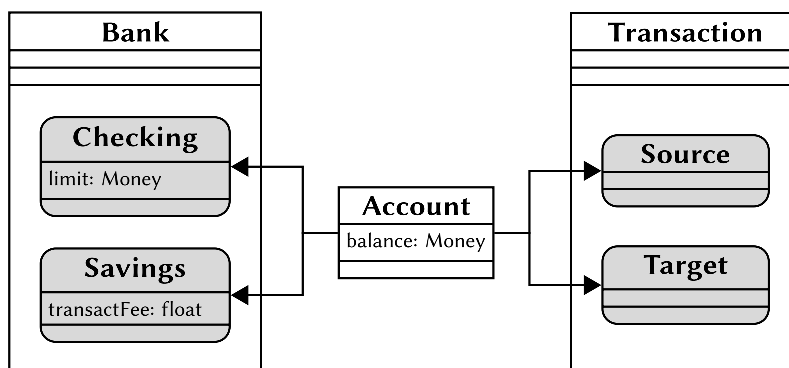


Figure 5.2: Role model for the banking example in CROM notation

Savings, Source and Target. In the following examples, we successively model a money transfer from one Account to another where the first one plays the Source role and the second one plays the Target role.

5.2.1 Role-based automata and their composition

RBA are an MDP-like formalism with additional annotations to capture role-specific behavior as well as the effects of binding a role to a player. For defining a notion of composability, we first introduce role interfaces that specify which roles appearing in an RBA are already bound to a player (the *bound roles*) and which roles are not bound yet (the *unbound roles*). Note that role interfaces always refer to role instances.

Definition 5.1 (Role interface)**.** A *role interface* is a pair $R = \langle B, U \rangle$ of a set of bound roles B and a set of unbound roles U with $B \cap U = \varnothing$.

We use the shorthand notation R for $B \cup U$ in case we do not explicitly refer to R as role interface. If $U = \varnothing$, then the role interface is *closed.* Two role interfaces $\langle B_1, U_1 \rangle$ and $\langle B_2, U_2 \rangle$ are called *compatible* if they have no common roles, i.e., $(B_1 \cup U_1) \cap (B_2 \cup U_2) = \varnothing$. We define a commutative and associative composition operator $\oplus$ on compatible role interfaces where $\langle B_1, U_1 \rangle \oplus \langle B_2, U_2 \rangle = \big\langle B_1 \cup B_2, (U_1 \cup U_2) \setminus (B_1 \cup B_2) \big\rangle$.

A *role annotation* denotes which roles are played (or not played) on a transition. The set of role annotations over a set of roles R is defined as $\mathbb{A}(R) = \{r, \overline{r}, +r : r \in R\}$. If a transition is labeled with r or $\overline{r}$, then the role r is actively played or explicitly not played, respectively. In case neither r nor $\overline{r}$ appear on a transition, then role r may be played, but not necessarily. Transitions annotated with $+r$ are added to the player upon binding the role r.

Definition 5.2 (Role-based automaton)**.** An RBA is a tuple $\mathcal{A} = (S, Act, R, \longrightarrow, S^{init})$, where

- S is a finite set of states,
- Act is a set of actions,
- $R = \langle B, U \rangle$ is a role interface,
- $\longrightarrow \subseteq S \times Act \times \mathcal{P}(\mathbb{A}(R)) \times Distr(S)$ is a role-annotated transition relation, and
- $S^{init} \subseteq S$ is a set of initial states.

A transition of a role-based automaton has the form (s, α, X, λ), where $s \in S$ is the source state, $\alpha \in Act$ is an action, $X \subseteq \mathbb{A}(R)$ is a set of role annotations, and λ is a distribution over S representing an internal probabilistic choice that specifies the probabilities for entering the successor states. We assume that transitions are not labeled with conflicting annotations, i.e., for all transitions and roles $r \in R$ we have $|X \cap \{r, \overline{r}, +r\}| \leq 1$. Further, we introduce the notation $s \xrightarrow{\alpha/X} \lambda$ for $(s, \alpha, X, \lambda) \in \longrightarrow$.

RBA provide a uniform representation for naturals, roles, and compartments. An RBA represents a natural if its role interface $R = \langle B, U \rangle$ is empty, a role if $B = \emptyset$ and U is a singleton, and a compartment otherwise.

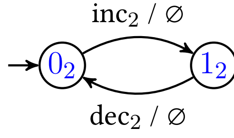


Figure 5.3: RBA for Account instance Acc_2

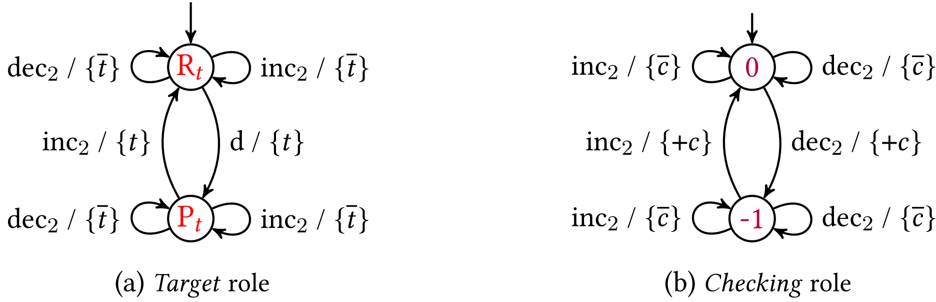


Figure 5.4: RBA for the Target role and the Checking role

Example 5.3 (Role-based automata for naturals and roles)**.** Figure 5.3 shows the RBA for the instance Acc_2 of the natural type Account. Since there are multiple accounts in the banking scenario, all states and actions of the presented RBA carry the index 2. To keep the example automata small, the account balance is either 0 or 1 and can be increased or decreased using the *inc* and *dec* actions, respectively.

In Figure 5.4a, the RBA representing an instance of the Target role is shown. Note that the name *Target* refers to the RBA itself, while t is the role's name in the role interface. In the *ready* state (R_t), the deposit action d may be invoked which enters the processing state (P_t). Then, the account balance is increased by synchronizing with the player's inc_2 action. The self-loops on both states annotated with $\{\bar{t}\}$ enable the inc_2 and dec_2 actions in case the role is not played. The Source role can be modeled analogously.

The RBA for an instance of the Checking account role is presented in Figure 5.4b. This role enables an account to be overdrawn. For the sake of simplicity, the account can only be overdrawn once, indicated by the state (-1). The $\{+c\}$ role annotations allow the inc_2 and dec_2 actions to be taken even if the player of the role does not provide these actions in its current state. This effectively enables overdrawing an empty account via the dec_2 action. ■

We call an RBA an *unbound role instance* of a role r if each transition is annotated with either r, $\bar{r}$, or $+r$, and $R = \langle \emptyset, \{r\} \rangle$. Note that the RBA in Figures 5.4a and 5.4b are unbound role instances of the Target role t and Checking account role c, respectively.

Parallel composition

Having established RBA as basic building blocks of the modeling formalism, we turn to the definition of the parallel composition operator which formalizes the interaction between RBA. The definition of the operator follows the usual definition of parallel composition for MDPs. Two automata in parallel run concurrently and can communicate via handshaking, i.e., synchronization over shared actions.

Definition 5.4 (Parallel composition)**.** The *parallel composition* of two RBA $\mathcal{A}_i = (S_i, Act_i, R_i, \longrightarrow_i, S_i^{init})$ with compatible role interfaces R_i for $i \in \{1, 2\}$ is defined as

$$\mathcal{A}_1 \parallel \mathcal{A}_2 \;=\; \big(S_1 \times S_2,\; Act_1 \cup Act_2,\; R_1 \oplus R_2,\; \longrightarrow,\; S_1^{init} \times S_2^{init}\big)$$

where $\longrightarrow$ is the smallest transition relation fulfilling the rules shown in Figure 5.5.

$$(\mathrm{int}_1)\frac{s_1 \xrightarrow{\alpha/X}_1 \lambda_1 \qquad \alpha \in Act_1 \setminus Act_2}{\langle s_1, s_2\rangle \xrightarrow{\alpha/X} \lambda_1 * Dirac(s_2)} \qquad (\mathrm{int}_2)\frac{s_2 \xrightarrow{\alpha/Y}_2 \lambda_2 \qquad \alpha \in Act_2 \setminus Act_1}{\langle s_1, s_2\rangle \xrightarrow{\alpha/Y} Dirac(s_1) * \lambda_2}$$

$$(\mathrm{sync})\frac{s_1 \xrightarrow{\alpha/X}_1 \lambda_1 \qquad s_2 \xrightarrow{\alpha/Y}_2 \lambda_2 \qquad \alpha \in Act_1 \cap Act_2}{\langle s_1, s_2\rangle \xrightarrow{\alpha/X \cup Y} \lambda_1 * \lambda_2}$$

Figure 5.5: Rules for the parallel composition of RBA

The parallel composition is well-defined, i.e., composing two RBA yields again an RBA. Note that due to the union of role annotations in the (sync) rule of Figure 5.5, multiple roles can be played at the same time in a single transition.

Role-binding

Next, we consider the role-binding operator. Role-binding joins the behavior of a role with that of its player, possibly modifying the player's behavior, such that the player is able to actively play the bound role. Existing approaches for role-binding presented in [FK01; TK03a; TK03b] and the closely related aspect weaving presented in [KFG04] realize composition by taking the union of the components' state spaces. This effectively results in a sequential execution of player behavior and role behavior. In contrast to these approaches, we employ a product construction similar to parallel composition. The product construction enables more flexibility for modeling the joint role-and-player behavior, as it allows a concurrent execution of role behavior and player behavior. Note that the sequential composition is fully covered by the product construction. By using

overriding transitions to switch into the role behavior and subsequently blocking all player behavior, a sequential execution be achieved.

Definition 5.5 (Role-binding)**.** Let $\mathcal{A} = (S_a, Act_a, R_a, \longrightarrow_a, S_a^{init})$ and $\mathcal{P} = (S_p, Act_p, R_p, \longrightarrow_p, S_p^{init})$ be RBA with compatible role interfaces $R_a = \langle B_a, U_a \rangle$ and R_p. *Binding* an unbound role $r \in U_a$ in $\mathcal{A}$ to a player $\mathcal{P}$ yields an RBA

$$\mathcal{A}[r \to \mathcal{P}] \;=\; \big(S_a \times S_p,\ Act_a \cup Act_p,\ R,\ \longrightarrow,\ S_a^{init} \times S_p^{init}\big)$$

where $R = R_a \oplus R_p \oplus \langle \{r\}, \varnothing \rangle$ and $\longrightarrow$ is the smallest transition relation fulfilling the rules shown in Figure 5.6.

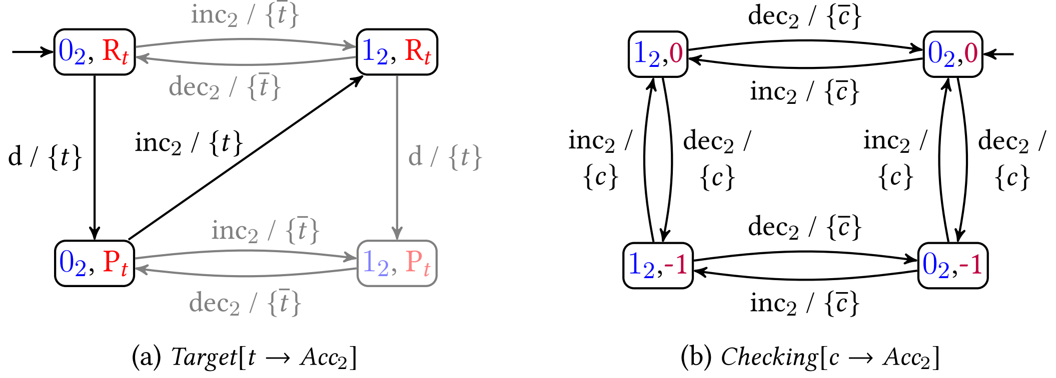
$$(\text{int}_a)\ \frac{s_a \xrightarrow{\alpha/X}_a \lambda_a \qquad \alpha \in Act_a \setminus Act_p}{\langle s_a, s_p \rangle \xrightarrow{\alpha/X} \lambda_a * Dirac(s_p)} \qquad (\text{int}_p)\ \frac{s_p \xrightarrow{\alpha/Y}_p \lambda_p \qquad \alpha \in Act_p \setminus Act_a}{\langle s_a, s_p \rangle \xrightarrow{\alpha/Y} Dirac(s_a) * \lambda_p}$$

$$(\text{sync})\ \frac{s_a \xrightarrow{\alpha/X}_a \lambda_a \qquad s_p \xrightarrow{\alpha/Y}_p \lambda_p \qquad \alpha \in Act_a \cap Act_p \qquad +r \notin X}{\langle s_a, s_p \rangle \xrightarrow{\alpha/X \cup Y} \lambda_a * \lambda_p}$$

$$(\text{add})\ \frac{s_a \xrightarrow{\alpha/X}_a \lambda_a \qquad \alpha \in Act_a \cap Act_p \qquad +r \in X}{\langle s_a, s_p \rangle \xrightarrow{\alpha/X \setminus \{+r\} \cup \{r\}} \lambda_a * Dirac(s_p)}$$

Figure 5.6: Rules for binding a role r within an RBA $\mathcal{A}$ to a player $\mathcal{P}$

Note that the interleaving rules in Figure 5.6 are defined as for parallel composition. The (sync) rule contains the additional precondition that the role transition is not annotated with $+r$. Rule (add) covers the case where r adds and possibly overrides an action of the player to which r is bound, without any synchronization with the player. This effectively allows the role to *add* new behavior to its player using the role annotation $+r$. Note that a role can also *suppress* transitions of the player by simply blocking them, i.e., by not providing a matching synchronizing transition. Combining both effects, adding and suppressing, allows the role to *change* the behavior of the player. In this case, we say that the role *overrides* the player's behavior. In particular, overriding happens if both the role and the player provide transitions that are labeled with the same action α and the role transition is annotated with $+r$. Then, the role will take its transition alone while the player remains in the same state (if the role is actively played). Therefore, the player's transition is effectively replaced by the role transition.



(a) $Target[t \to Acc_2]$ (b) $Checking[c \to Acc_2]$

Figure 5.7: Resulting RBA for binding the *Target* role and the *Checking* role to Acc_2

Example 5.6 (Role-binding)**.** For the banking running example, the result of binding the *Target* role (Figure 5.3) to the account Acc_2 (Figure 5.4a) is shown in Figure 5.7a. The transitions where the *Target* role t is actively played to receive a deposit are emphasized.

Figure 5.7b shows the result of binding the *Checking* role (Figure 5.4b) to the account Acc_2. The *Checking* role (see Figure 5.4b) provides a transition labeled with action dec_2 (from state (0) to (-1)). Since this transition is annotated with $\{+c\}$, it gets *added* to the player upon binding which enables overdrawing the account, subsequently moving from state $(0_2,0)$ to $(0_2,\text{-}1)$. An example for *overriding* is the transition from state $(1_2,0)$ to $(1_2,\text{-}1)$. Here, the dec_2 transition of Acc_2 is blocked and then replaced by the transition of the *Checking* role.

It is possible to bind multiple roles to the same player via nested role-binding. For instance, binding both the *Checking* role and the *Target* role to the same account Acc_2 can be achieved by a composition $Target[t \to Checking[c \to Acc_2]]$. ■

Let us discuss again the distinction between role-binding and role-playing. As mentioned previously, binding a role enables a player to act in that role. The ability to play a role is a *state* property. If a state has outgoing transitions annotated with r, then the role r can be played. Role-playing, on the other hand, happens within *transitions*. A role r is actively played while taking a transition that is annotated with r. Therefore, the distinct notions of a role being bound and a role being played are clearly reflected in the structure of an RBA.

Non-blocking roles

It follows from the (sync) rule in Figure 5.6 that a bound role can block actions of the player even if the role is not actively played. More precisely, this is the case when the player would be able to perform a transition labeled with action α, but a bound and not played role r does not provide an α transition with role annotation $\bar{r}$. We call an RBA for role r *non-blocking for an action* α if every state s of the associated RBA has a self-loop

$(s, \alpha, \{\bar{r}\}, Dirac(s))$. Further, we call a role RBA $\mathcal{A}$ *non-blocking* w.r.t. to a player $\mathcal{P}$ in case the role is non-blocking for all actions $\alpha \in Act_{\mathcal{A}} \cap Act_{\mathcal{P}}$, where $Act_{\mathcal{A}}$ and $Act_{\mathcal{P}}$ are the action sets of the role RBA $\mathcal{A}$ and the player RBA $\mathcal{P}$, respectively. If an RBA $\mathcal{A}$ for role r is non-blocking w.r.t. a player $\mathcal{P}$, then the player's original behavior is preserved upon role-binding. That is, if the role r is not actively played, i.e., no transitions annotated with r are taken, then the RBA $\mathcal{A}[r \to \mathcal{P}]$ behaves exactly the same as $\mathcal{P}$. The roles shown in Figure 5.4 are indeed non-blocking for all actions of the account Acc_2 and the original behavior of Acc_2 is preserved after role-binding (see the horizontal transitions in Figure 5.7). Blocked behavior caused by role-binding is often unintuitive and leads to easily overseen side-effects. Thus, the implementation of the approach presented here (see Chapter 6) follows other role-oriented programming languages and preserves the role players' original behavior in case their roles are not actively played. This is achieved by automatically transforming each role into a non-blocking role w.r.t. its player.

5.2.2 Algebraic properties of compositions

In this section, we discuss the interactions between the composition operators on RBA defined in the previous section. The parallel composition and role-binding operators fulfill certain algebraic properties, highlighted in the following theorem.

Theorem 5.7 (Algebraic properties of compositions). *For pairwise compatible RBA $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{P}_1, \mathcal{P}_2$ we have*

$$\mathcal{A}_1 \parallel \mathcal{A}_2 \cong \mathcal{A}_2 \parallel \mathcal{A}_1 \tag{5.1}$$

$$\mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3) \cong (\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3 \tag{5.2}$$

If it additionally holds that $\mathcal{A}_1$ and $\mathcal{A}_2$ are unbound role instances with names a_1 and a_2, respectively, we have

$$(\mathcal{A}_1 \parallel \mathcal{A}_2)[a_1 \to \mathcal{P}_1] \cong \mathcal{A}_1[a_1 \to \mathcal{P}_1] \parallel \mathcal{A}_2 \tag{5.3}$$

$$(\mathcal{A}_1 \parallel \mathcal{A}_2)[a_1 \to \mathcal{P}_1][a_2 \to \mathcal{P}_2] \cong (\mathcal{A}_1 \parallel \mathcal{A}_2)[a_2 \to \mathcal{P}_2][a_1 \to \mathcal{P}_1] \tag{5.4}$$

where $\cong$ stands for isomorphism (equality up to renaming of states). If $Act_1 \cap Act_2 = \emptyset$ for the action sets Act_1 and Act_2 of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, it further holds that

$$\mathcal{A}_1[a_1 \to \mathcal{A}_2[a_2 \to \mathcal{P}_1]] \cong \mathcal{A}_2[a_2 \to \mathcal{A}_1[a_1 \to \mathcal{P}_1]] \tag{5.5}$$

Proof. Let $\mathcal{A}_i = (S_i, Act_i, R_i, \longrightarrow_i, S_i^{init})$ and $\mathcal{P}_j = (S_{\mathcal{P}_j}, Act_{\mathcal{P}_j}, R_{\mathcal{P}_j}, \longrightarrow_{\mathcal{P}_j}, S_{\mathcal{P}_j}^{init})$ be RBA for $i \in \{1, 2, 3\}$ and $j \in \{1, 2\}$.

eq. (5.1): Commutativity of the parallel composition is clear since $\cup$ and $*$ (the product measure on distributions) are both commutative. Furthermore, the conditions in

the premise of (sync) are symmetric and (int_1) is symmetric to (int_2). Compatibility of the role interfaces is also clear by the commutativity of interface compatibility.

eq. (5.2): Associativity of the parallel composition for RBA follows directly from the associativity of the parallel composition for MDPs. The only difference are the additional role-playing annotations. However, associativity is also given here by the associativity of $\cup$. It remains to show compatibility of the role interfaces. From the left-hand side, we obtain $R_2 \cap R_3 = \emptyset$ and $R_1 \cap (R_2 \cup R_3) = \emptyset$. Thus, $R_1 \cap R_2 = \emptyset$ and $R_1 \cap R_3 = \emptyset$. Therefore, $(R_1 \cup R_2) \cap R_3 = \emptyset$.

eq. (5.3): For proving this property, we rely on the associativity of the parallel composition (eq. (5.2)). Note that the rules for role-binding correspond to the rules for parallel composition in case a transition is not labeled with a $+r$ annotation. Thus, it remains to show that eq. (5.3) holds in case $\mathcal{A}_1$ has transitions labeled with $+a_1$. Assume there is a transition $s_1 \xrightarrow{\alpha/X \cup \{+a_1\}}_1 \lambda_1$. We have to consider the cases where $\alpha \in Act_1 \setminus Act_2$ (int_1) and $\alpha \in Act_1 \cap Act_2$ (sync).

(int_1) Applying rule (add) on the left-hand side yields for all $s_2 \in S_2$ and $s_{\mathcal{P}_1} \in S_{\mathcal{P}_1}$ transitions

$$\langle s_1, s_2, s_{\mathcal{P}_1} \rangle \xrightarrow{\alpha/X \setminus \{+a_1\} \cup \{a_1\}} \lambda_1 * Dirac(s_2) * Dirac(s_{\mathcal{P}_1}) \ .$$

On the right-hand side, binding a_1 in $\mathcal{A}_1$ to $\mathcal{P}_1$ yields for all $s_{\mathcal{P}_1} \in S_{\mathcal{P}_1}$ the transitions

$$\langle s_1, s_{\mathcal{P}_1} \rangle \xrightarrow{\alpha/X \setminus \{+a_1\} \cup \{a_1\}} \lambda_1 * Dirac(s_{\mathcal{P}_1}) \ .$$

Then, the parallel composition with $\mathcal{A}_2$ yields

$$\langle s_1, s_{\mathcal{P}_1}, s_2 \rangle \xrightarrow{\alpha/X \setminus \{+a_1\} \cup \{a_1\}} \lambda_1 * Dirac(s_{\mathcal{P}_1}) * Dirac(s_2) \ .$$

(sync) Applying rule (add) on the left-hand side yields for all $s_{\mathcal{P}_1} \in S_{\mathcal{P}_1}$ transitions

$$\langle s_1, s_2, s_{\mathcal{P}_1} \rangle \xrightarrow{\alpha/X \setminus \{+a_1\} \cup \{a_1\}} \lambda_1 * \lambda_2 * Dirac(s_{\mathcal{P}_1}) \ .$$

On the right-hand side, binding a_1 in $\mathcal{A}_1$ to $\mathcal{P}_1$ yields for all $s_{\mathcal{P}_1} \in S_{\mathcal{P}_1}$ the transitions

$$\langle s_1, s_{\mathcal{P}_1} \rangle \xrightarrow{\alpha/X \setminus \{+a_1\} \cup \{a_1\}} \lambda_1 * Dirac(s_{\mathcal{P}_1}) \ .$$

Then, the parallel composition with $\mathcal{A}_2$ yields

$$\langle s_1, s_{\mathcal{P}_1}, s_2 \rangle \xrightarrow{\alpha/X \setminus \{+a_1\} \cup \{a_1\}} \lambda_1 * Dirac(s_{\mathcal{P}_1}) * \lambda_2 \ .$$

eq. (5.4): For proving this property, we rely on the commutativity of the parallel composition (eq. (5.1)). Remind that the rules for role-binding correspond to the rules for parallel composition in case a transition is not labeled with a $+r$ annotation. Commutativity of the (add) rule is clear since $(X \setminus \{+r_1\} \cup \{r_1\}) \setminus \{+r_2\} \cup \{r_2\} = (X \setminus \{+r_2\} \cup \{r_2\}) \setminus \{+r_1\} \cup \{r_1\}$.

eq. (5.5): For the rules (int$_a$), (int$_p$), and (sync) property 5.5 is clear by associativity (eq. (5.2)) and commutativity (eq. (5.1)) of the parallel composition. It remains to show that eq. (5.5) holds if $\mathcal{A}_1$ or $\mathcal{A}_2$ have overriding transitions. Assume $\mathcal{A}_1$ has a transition $s_1 \xrightarrow{\alpha/X\cup\{+a_1\}} \lambda_1$ with $\alpha \in Act_1 \cap Act_{\mathcal{P}_1}$. Then, binding $\mathcal{A}_1$ on the left-hand side yields for all $s_2 \in S_2$ and $s_{\mathcal{P}_1} \in S_{\mathcal{P}_1}$ transitions

$$\langle s_1, s_2, s_{\mathcal{P}_1} \rangle \xrightarrow{\alpha/X\setminus\{+a_1\}\cup\{a_1\}} \lambda_1 * Dirac(s_2) * Dirac(s_{\mathcal{P}_1}) \ .$$

On the right-hand side, binding $\mathcal{A}_1$ yields for all $s_{\mathcal{P}_1} \in S_{\mathcal{P}_1}$ transitions

$$\langle s_1, s_{\mathcal{P}_1} \rangle \xrightarrow{\alpha/X\setminus\{+a_1\}\cup\{a_1\}} \lambda_1 * Dirac(s_{\mathcal{P}_1}) \ .$$

Since $Act_1 \cap Act_2 = \emptyset$, there is no synchronization of $\mathcal{A}_2$'s transitions with the above transitions introduced by binding $\mathcal{A}_1$. Thus, binding $\mathcal{A}_2$ yields for all $s_2 \in S_2$ and $s_{\mathcal{P}_1} \in S_{\mathcal{P}_1}$ transitions

$$\langle s_2, s_1, s_{\mathcal{P}_1} \rangle \xrightarrow{\alpha/X\setminus\{+a_1\}\cup\{a_1\}} Dirac(s_2) * \lambda_1 * Dirac(s_{\mathcal{P}_1}) \ .$$

The case where $\mathcal{A}_2$ has an overriding transition can be shown as for $\mathcal{A}_1$ by swapping $\mathcal{A}_1$ and $\mathcal{A}_2$.

□

Equations (5.1) and (5.2) state that the parallel composition of RBA is both commutative and associative. From eq. (5.3), we obtain that role-binding can be applied both before and after the parallel composition of roles. Furthermore, role-binding is commutative if the roles are bound to different players (eq. (5.4)). Additionally, binding two roles to the same player also commutes in case the roles are *independent*, i.e., the action sets of the roles' RBA are disjoint, as stated in eq. (5.5). The latter property allow us to bind multiple roles to the same player using nested role-binding, and in turn, allows the player to act in multiple compartments at the same time.

The commutativity of binding two roles to the same player cannot be expected if the roles are not independent, i.e., their action sets are not disjoint. Since role-binding allows the replacement of transitions (cf. Definition 5.5), the order in which the roles are bound is crucial. An example is given in Figure 5.8. Here, the role $\mathcal{A}_1$ synchronizes with its player over the α action, while $\mathcal{A}_2$ overrides the α transition of its player. If $\mathcal{A}_1$ is bound last (see

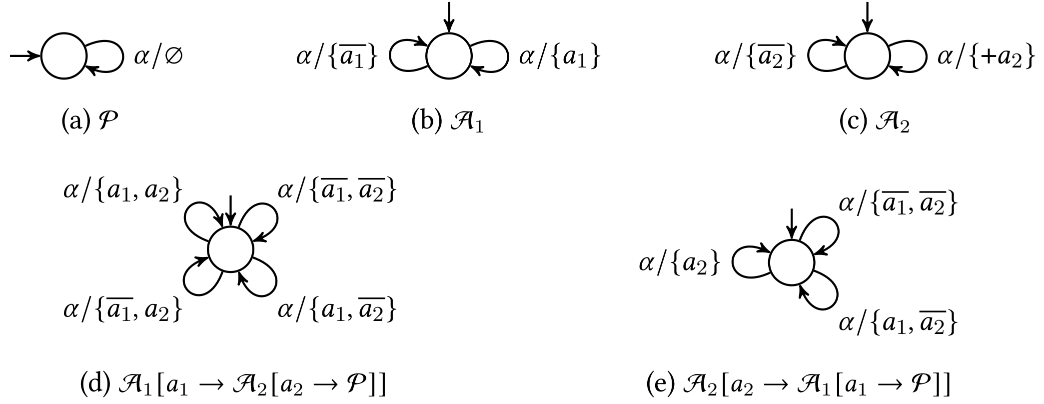


Figure 5.8: Example showing that binding two roles to the same player does not commute in case the roles' action sets are not disjoint

Figure 5.8d), it can synchronize with both $\mathcal{A}_2$ and $\mathcal{P}$, resulting in the transition annotated with $\{a_1, a_2\}$. This transition does not exist if $\mathcal{A}_2$ is bound last, since $\mathcal{A}_2$ overrides the α transition of $\mathcal{A}_1$. The ability to bind roles which modify not only the behavior of the player but also the behavior of other roles is often useful for the separation of concerns. In the banking scenario for instance, the *Target* role (Figure 5.4a) can be bound to an account that already has the *Checking* role (Figure 5.4b) bound to it. Then, the account can be overdrawn in a money transfer, while the concerns of overdrawing and withdrawing are still cleanly separated into the *Checking* and *Target* roles, respectively.

Not only the order in which roles are bound to a player is important, but also the order in which the binding operator itself is applied can affect the final result. Suppose there are two roles o and i with associated RBA $\mathcal{A}_o$ and $\mathcal{A}_i$, respectively, that shall be bound to the same player RBA $\mathcal{P}$. For that, we can first bind the role i to the player, followed by the binding of the role o, formally $\mathcal{A}_o[o \to \mathcal{A}_i[i \to \mathcal{P}]]$. However, we could also bind the role o first (to the role i) and then bind the role i to the player, formally $\big(\mathcal{A}_o[o \to \mathcal{A}_i]\big)[i \to \mathcal{P}]$. Figure 5.9 shows an example where the different binding orders result in different RBA. If the role o is bound first, then its overriding transition (annotated with $\{+o\}$) only affects the RBA $\mathcal{A}_i$, but not $\mathcal{P}$. However, if the role o is bound last, then the overriding transition applies to both $\mathcal{A}_i$ and $\mathcal{P}$. Note that in general $\mathcal{A}_o[o \to \mathcal{A}_i[i \to \mathcal{P}]] \ncong \big(\mathcal{A}_o[o \to \mathcal{A}_i]\big)[i \to \mathcal{P}]$ even in case $Act_o \cap Act_i = \emptyset$.

5.2.3 Coordination and semantics of RBA

We define the semantics of an RBA with respect to a coordination component called *role-playing coordinator*. The composition of an RBA with a coordinator resolves all role-playing and yields an MDP where the role-playing is encoded into the action labels.

A role annotation (as defined in Section 5.2.1) is a symbolic representation that may stand for multiple role-playings. We define a *role-playing* as a set $\mathcal{I} \subseteq R$ that contains all

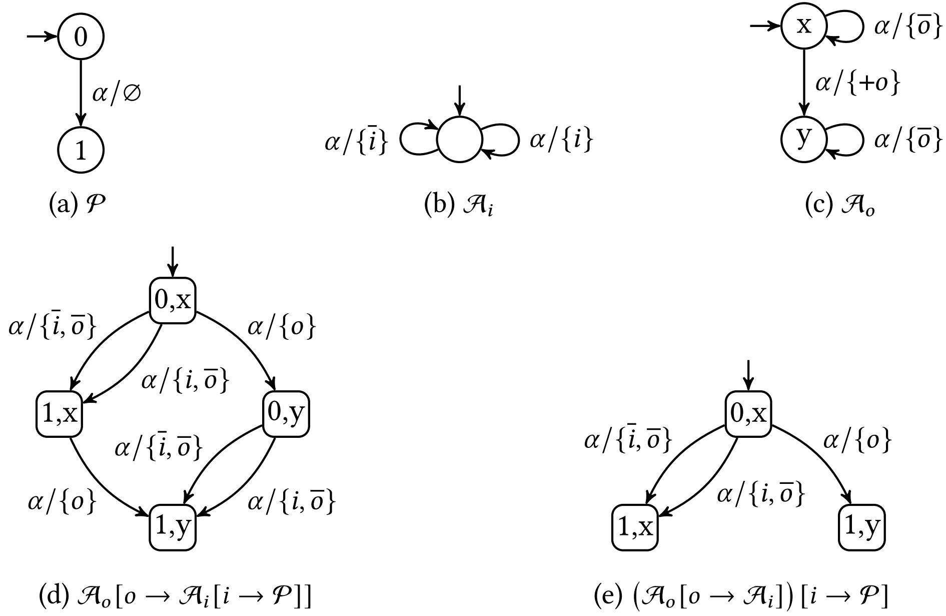


Figure 5.9: Example showing that the order in which role-binding is applied affects the result

roles that are played within a transition, i.e., if $r \in R$ and $r \in \mathcal{I}$ then r is explicitly played, and if $r \notin \mathcal{I}$ then r is not played. We further define the set of *possible role-playings* as the set of all role-playings permitted by a role annotation.

Definition 5.8 (Possible role-playings)**.** The set of *possible role-playings* $\mathbb{R}(X, R) \subseteq \mathcal{P}(R)$ with respect to a role annotation $X \in \mathbb{A}(R)$ comprises exactly those $\mathcal{I} \subseteq R$ for which

(1) for all $r \in \mathcal{I}$ we have $\overline{r} \notin X$, and
(2) for all $r \in R$ with $\{r, +r\} \cap X \neq \varnothing$ we have $r \in \mathcal{I}$.

Intuitively, for any $\mathcal{I} \in \mathbb{R}(X, R)$ the rules above state that $\mathcal{I}$ does not contain a role that must not be played (1) and does contain all roles that are played (2). The role annotation X in a transition (s, α, X, λ) of an RBA over R intuitively means that all combinations of roles $\mathcal{I} \in \mathbb{R}(X, R)$ fulfilling the constraints imposed by X can be played within the transition.

Role-playing coordinator

The *role-playing coordinator* specifies the rules for role-playing in an RBA. Essentially, a role-playing coordinator is an RBA with a different semantics.

Definition 5.9 (Coordinator)**.** A (role-playing) *coordinator* is an RBA $\mathrm{C} = (S, Act, R, \longrightarrow, S^{init})$ for which R is closed and for all transitions $(\ell, \alpha, Y, \lambda) \in \longrightarrow$ and $r \in R$ we have $+r \notin Y$.

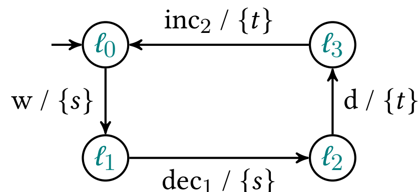


Figure 5.10: A role-playing coordinator for the banking scenario coordinating money transfers

The role-playing coordinator represents a unified formalism to specify both static constraints and temporal constraints on role-playing. A static constraint must hold at all times, while a temporal constraint imposes some order on the role-playing. Additionally, the coordinator may "implement" a context model that associates a specific context to each environment or situation the system may encounter. Upon context changes, the coordinator then activates and deactivates the appropriate roles. Note that since the coordinator is basically an RBA, the parallel composition on RBA can also be applied to combine multiple coordinators. This enables a separation of concerns on the coordination level by, e.g., assigning a dedicated coordinator to each compartment.

Example 5.10 (Money-transfer coordinator)**.** Figure 5.10 shows the role-playing coordinator for money transfers in the banking scenario. First, money is withdrawn from account 1 using the *Source* role s which *dec*reases the account's balance. Then, the transfered money is deposited on account 2 via the *Target* role t, which subsequently *inc*reases the balance. This coordinator specifies a temporal constraint, namely that the *Source* role s must be played before *Target* role t. Additionally, it defines a static constraint implicitly. The roles s and t cannot be played simultaneously at any time, since there is no coordinator transition annotated with $\{s, t\}$. ■

As mentioned previously, we distinguish between role-binding and role-playing. The role-playing coordinator is only concerned with playing which raises the question whether a similar construct for role-binding exists, i.e., a coordination component that can dynamically bind and unbind roles. For answering that question, we first have to clarify what it means to unbind a role. Unbinding is the inverse of binding, thus after unbinding a role the player should behave as before the role has been bound. Thus, in order to be able to unbind a role, the original behavior of the player must be preserved. As discussed in Section 5.2.1, the player's behavior is preserved as long as all its roles are non-blocking w.r.t. the player. Therefore, if all roles are non-blocking, then the effect of dynamic unbinding can be achieved using the role-playing coordinator by only allowing transitions in which the "unbound" role is not played.

Semantics of RBA

Given an RBA $\mathcal{A}$ that arises from the composition of naturals, roles, and compartments, and a coordinator C formalizing the rules for role-playing, the operational semantics of $\mathcal{A}$ under C is an MDP defined as follows.

Definition 5.11 (MDP semantics)**.** Composing an RBA $\mathcal{A} = (S_a, Act_a, R_a, \longrightarrow_a, S_a^{init})$ and a coordinator $\mathrm{C} = (S_c, Act_c, R_c, \longrightarrow_c, S_c^{init})$ yields an MDP

$$[\![\mathcal{A}]\!]_{\mathrm{C}} \;=\; \big(\, S_a \times S_c,\ Act,\ \longrightarrow,\ S_a^{init} \times S_c^{init} \,\big)$$

where $Act = (Act_a \cup Act_c) \times \mathcal{P}(R)$ with $R = R_a \cup R_c$, and $\longrightarrow \subseteq S \times Act \times Distr(S)$ is the smallest transition relation fulfilling the rules shown in Figure 5.11

$$(\mathrm{int}_a)\ \frac{s \xrightarrow{\alpha/X}_a \lambda \qquad \alpha \in Act_a \backslash Act_c \qquad \mathcal{I} \in \mathbb{R}(X, R)}{\langle s, \ell\rangle \xrightarrow{\langle \alpha, \mathcal{I}\rangle} \lambda * Dirac(\ell)} \qquad (\mathrm{int}_c)\ \frac{\ell \xrightarrow{\alpha/Y}_c \lambda_c \qquad \alpha \in Act_c \backslash Act_a \qquad \mathcal{I} \in \mathbb{R}(Y, R)}{\langle s, \ell\rangle \xrightarrow{\langle \alpha, \mathcal{I}\rangle} Dirac(s) * \lambda_c}$$

$$(\mathrm{sync})\ \frac{s \xrightarrow{\alpha/X}_a \lambda_s \qquad \ell \xrightarrow{\alpha/Y}_c \lambda_c \qquad \alpha \in Act_a \cap Act_c \qquad \mathcal{I} \in \mathbb{R}(X \cup Y, R)}{\langle s, \ell\rangle \xrightarrow{\langle \alpha, \mathcal{I}\rangle} \lambda_s * \lambda_c}$$

Figure 5.11: Rules for the composition of an RBA and a coordinator

The interaction between the RBA and the coordinator is formalized by synchronization over both actions and role-playing. This allows the coordinator to enforce and restrict role-playing by providing only the appropriate transitions. Additionally, the coordinator may monitor and react to role-playing which enables the specification of temporal constraints. For a trivial coordinator consisting of only a single state and an empty transition relation, the MDP-semantics for RBA arises from resolving each possible role-playing nondeterministically due to (int_a) in Figure 5.11 being the only applicable rule.

Example 5.12 (MDP for the money transfer)**.** Figure 5.12 shows the MDP for the money transfer in the banking scenario with two accounts, the first one playing the *Source* role and the second one playing the *Target* role under the coordinator shown in Figure 5.10. The system works as specified, as money is withdrawn from the source account (by decrementing its balance) and then deposited on the target account. ■

Having MDPs as underlying semantics for role-based systems allows us to apply methods for simulation, verification, and quantitative analysis, such as standard model-checking algorithms for temporal logics, e.g., LTL and PCTL. To reason about role-playing, which is encoded into the action labels of the resulting MDP, standard approaches

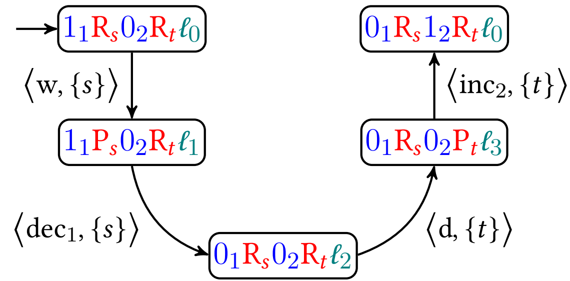


Figure 5.12: Resulting MDP of the composition $[\![\mathit{Source}[s \rightarrow \mathit{Acc}_1] \parallel \mathit{Target}[t \rightarrow \mathit{Acc}_2]]\!]_{\mathrm{C}}$

using action-based logics may be employed [DV90]. The implementation of a role-oriented modeling language presented in the next chapter leverages the MDP-semantics by providing a translation of a high-level modeling language into the input language of the model checker PRISM.

6 Implementation of a role-oriented modeling language

In this chapter, an implementation for the formal modeling and analysis of role-based systems is presented. It is based on the modeling formalism described in Chapter 5. Since using RBA directly for modeling is infeasible for larger systems, the implementation provides a role-oriented modeling language (RML) with RBA-semantics for concisely specifying RBA and role-playing coordinators. Leveraging the MDP-semantics of RBA under a coordinator, the tool RBSC translates a role-based model into the input language of the probabilistic model checker PRISM. This allows us to reuse existing infrastructure and PRISM's extensive analysis support. An overview of the approach is provided in Figure 6.1.

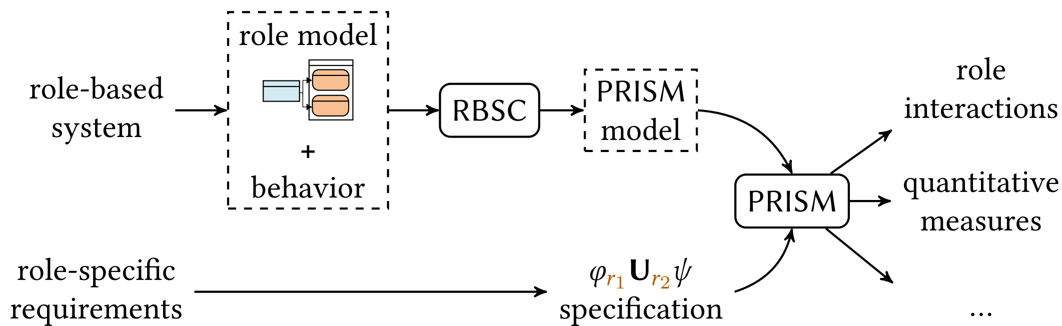


Figure 6.1: Overview of the approach

Outline. In Section 6.1, the role-oriented modeling language is introduced. Notable implementation details of the translation into PRISM's input language are described in Section 6.2. We continue to use the banking running example introduced in Section 5.2 to illustrate the concepts described in this chapter.

The implementation has been presented partially in the publication [Chr+20].

6.1 Role-oriented modeling language

A role-based model comprises two distinct parts: a declarative description of the system structure and a modular representation of the operational behavior for each component of

the system. In the following, we use the generic term *component* to refer to naturals, roles, and compartments. For describing the component structure of the system, we take inspiration from existing meta-models for role modeling, such as the CROM [Küh+15]. Since the modeling language is intended to be used in formal analysis and has an automata-based semantics, the component behavior is defined in terms of guarded commands [JSM97] where a command is a symbolic representation of possibly multiple transitions. The syntax is inspired by the input language of PRISM, thus our modeling language can be seen as a role-oriented extension of the PRISM language. In the following, the constructs of the modeling language RML are explained in detail, starting with the description of the system structure.

6.1.1 Declaration of the system structure

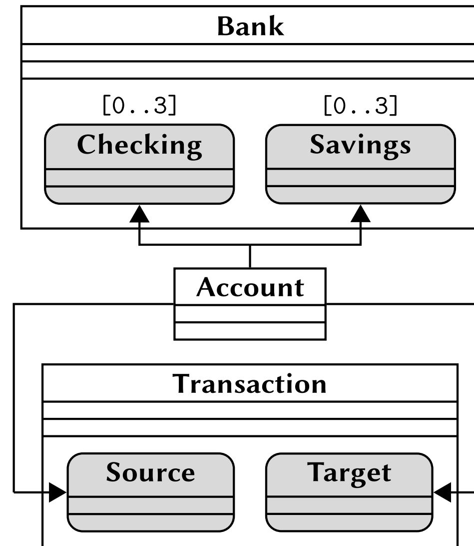
The modeling formalism presented in Section 5.2 solely concerns the instance level, i.e., every RBA represents a single component instance or a set of component instances. RML provides an additional type level which means that each component instance has an associated component type. This has two important implications for the modeling of role-based systems. First, component instances can be derived from a common representation by instantiation of a component type, comparable to the concept of creating an object from a class in object-oriented languages. Second, constraints regarding the system structure can not only be defined over individual instances, but also over component types, which allows us to formulate general constraints over sets of instances. In RML, the structure of a system is defined by a set of component types and a set of additional constraints that describe the structure to which instances of the role-based system must adhere to.

Component types

A model contains a type definition for each component type as exemplified in Listing 6.2a. Every definition of a natural type, role type, or compartment type declares a (unique) type name. Additionally, role-type definitions and compartment-type definitions specify general constraints that must hold for all instances of these types. In particular, a role-type definition is followed by a list of component types whose instances are allowed as players of the role (lines 3–9). Note that the player types are not limited to natural types, but can also include other role types or compartment types. Indeed, the `Source` and `Target` role types list both the roles `Checking` and `Savings` as possible players (lines 6–9). Subsequently, this allows us to bind both, e.g., a `Checking` role instance and a `Source` role instance to the same `Account` instance using nested role-binding. A compartment-type definition incorporates a list of role occurrence constraints. An instance of the compartment is required to contain exactly one role instance for each of the listed role types. Thus, both a `Source` role instance and a `Target` role instance

```
1  natural type Account;
2
3  role type Checking(Account);
4  role type Savings(Account);
5
6  role type Source(Account,
7    Checking, Savings);
8  role type Target(Account,
9    Checking, Savings);
10
11 compartment type Transaction(
12   Source, Target);
13
14 compartment type Bank(
15   Checking[0..3], Savings[0..3]);
```

(a) Component type definitions



(b) Corresponding role model

Figure 6.2: Type definitions for the banking scenario

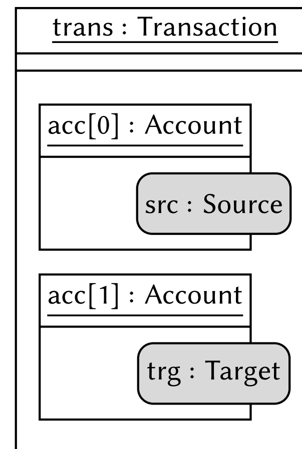
must be contained in an instance of the `Transaction` compartment type (lines 11–12). It is also possible to specify lower and upper bounds to refine the occurrence constraints, as shown in lines 14 and 15. An instance of the `Bank` compartment may contain up to 3 `Checking` roles and 3 `Savings` roles. Furthermore, multiple alternative lists of occurrence constraints can be specified. For instance, the definition `compartment type SmallBank(Checking | Savings)` states that a compartment instance must contain either a `Checking` or a `Savings` account, but not both. Alternatives and cardinalities can also be combined. The type definitions in RML can be seen as a textual representation of commonly used role models (such as the CROM) that describe the type level of a role-based system. Figure 6.2b shows the role model corresponding to the type definitions in the CROM's graphical notation. Note that the "fills" relation between the roles has been omitted from the graphical representation to reduce clutter.

Instantiating systems

After having defined the types the components of the role-based system may have, the next step is to actually create instances of these types in order to construct a concrete system. Generally speaking, systems are described by a set of constraints that define the set of component instances and specify how those components are related to each other. In particular, a constraint may specify which player is bound to a certain role instance or it may specify that a role instance belongs to a certain compartment. The constraint language is based on *first-order logic* (FOL) and interpreted over the set of component instances. It has three built-in predicates. The "`:`" (pronounced "has type")

```
system {
  acc[2] : Account;

  src : Source;
  trg : Target;

  trans : Transaction;

  src boundto acc[0];
  trg boundto acc[1];

  src in trans;
  trg in trans;
}
```

(a) System description



(b) Corresponding role model instance

Figure 6.3: Instantiation of a role-based system within the banking scenario

predicate states that the component instance on the left-hand side has the type given on the right-hand side. The second predicate, `boundto`, holds true if the role instance on the left-hand side is bound to the component on the right-hand side. Note that there are no restrictions regarding the player, except that a role cannot be bound to itself. Lastly, the `in` predicate states that a role instance is contained in a given compartment instance. The "`:`" predicate is special, in that it allows us to introduce new constant symbols. If `T` is a component type name and `c` is not defined elsewhere in the model, the constraint `c : T` introduces a new constant `c` which refers to a fresh component instance with type `T`. Furthermore, the construct `a[N] : T`, where `N` is an expression that evaluates to a non-negative integer, creates an array of component instances, all having type `T`. This construct is especially useful to generically define models that are parametrized by some size parameter. Together, the three built-in predicates are sufficient to fully describe an instance of a role-based system.

Example 6.1 (Instantiating a transaction compartment)**.** The constraints describing a system are given within a `system` block, as shown in Listing 6.3a where a transaction in the banking scenario is instantiated. In lines 2–7, the component instances of the system are listed, where the array instantiation construct is used to create two `Account` instances, `acc[0]` and `acc[1]`. The constraints in lines 9 and 10 state that the `Source` role instance `src` and the `Target` role instance `trg` are bound to `acc[0]` and `acc[1]`, respectively. Finally, the two role instances are placed into the `Transaction` compartment instance `trans` (lines 12–13). The corresponding system in the graphical CROI notation is shown in Figure 6.3b. ■

```
system {
  acc : Account;

  ch : Checking;
  src : Source;

  ch boundto acc;
  src boundto ch;
}
```

Listing 6.4: Binding multiple roles using nested binding

Example 6.2 (Nested role-binding)**.** As described in Section 5.2.1, multiple roles can be bound to the same player using nested role-binding. In the banking scenario, we may bind both a `Checking` role and a `Source` role to the same player, as shown in Listing 6.4. Note that the `Source` role `src` is not directly bound to the `Account` natural `acc`. This would be possible if the `src` role and the `Checking` role `ch` were independent, i.e., if their action alphabets were disjoint. However, this is not the case. Therefore, the order in which the roles are bound to their player is important. In RML, the binding order is derived from the graph induced by the **`boundto`** relation. Since the `ch` role is directly bound to the `acc` natural (line 7), it is bound first. The `src` role, on the other hand, is only indirectly bound to `acc`, in particular over the `ch` role (line 8). It is therefore bound second, to both the `ch` role and implicitly (because of the binding in line 7) also to the account `acc`. ■

A set of constraints is called *complete* if (1) each role instance is bound to a player, and (2) each compartment contains enough role instances to fulfill its occurrence constraints as defined by the compartment-type definition. The set of constraints in Listing 6.3a is complete, since both roles are bound to a player and the `trans` compartment instance contains exactly one `Source` and one `Target` role. A complete set of constraints corresponds to at most one system[1]. However, it is also possible to define multiple systems using a single **`system`** block by specifying a set of constraints which is not complete, i.e., where the players of some roles are unspecified or where not enough roles are assigned to some compartments. Generating multiple similar systems, i.e., a *system family*, using an incomplete set of constraints is useful for comparative studies and synthesis, e.g., for determining the best assignment of roles to players according to some optimization criterion.

The set of systems corresponding to an incomplete set of constraints is generated by an automated stepwise extension of the constraint set until the set is complete. There are possibly multiple different extensions at every step and every combination of extensions

[1] If the set of constraints is unsatisfiable, e.g., if there are contradicting constraints, then there is no corresponding system.

corresponds to a single system instance. Within a single step, the constraint set is extended as follows.

- If there is an unbound role instance `r` with role type `R`, the player of `r` is either created by instantiating one of the types listed in the definition of `R`, or a player with the right type is selected among the existing component instances. The role `r` is then bound to the instantiated or selected player.
- If there is a compartment instance `c` where, according to its compartment-type definition, the lower bound for some role type `R` is not satisfied, the missing role instance is either newly instantiated or selected among the existing role instances with type `R`. The instantiated or selected role is then assigned to the compartment `c`.

Note that the instantiation of a new player may create an unbound role or an empty compartment. Similarly, a newly created role within a compartment is initially unbound. In both cases, the newly created instances are also considered in the subsequent extension steps. Special care is taken to ensure that only a finite number of new instances is created. If within a single combination of extensions a type has been instantiated before, it is not instantiated again. The process of extending an incomplete set of constraints as outlined above ensures that all possible combinations of role-binding and compartment membership are explored while still only a minimal number of component instances is created.

```
1 natural type Account;
2 role type Checking(Account);
3 role type Savings(Account);
4 compartment type Bank(Checking, Savings);
5
6 system {
7   acc : Account;
8   bank : Bank;
9 }
```

Listing 6.5: Incomplete set of constraints for instantiating multiple systems

Example 6.3 (Systems corresponding to an incomplete constraint set)**.** Listing 6.5 comprises a simplified set of component-type definitions for the banking example. In particular, the `Bank` compartment must contain exactly one `Checking` role and one `Savings` role. The constraint set in the `system` block is incomplete, since there are no roles assigned to the `bank` compartment instance. Thus, the occurrence constraints of the `Bank` compartment type are not satisfied.

In order to generate the systems satisfying the constraints in Listing 6.5, the previously described extension process is applied. The decision tree containing the possible exten-

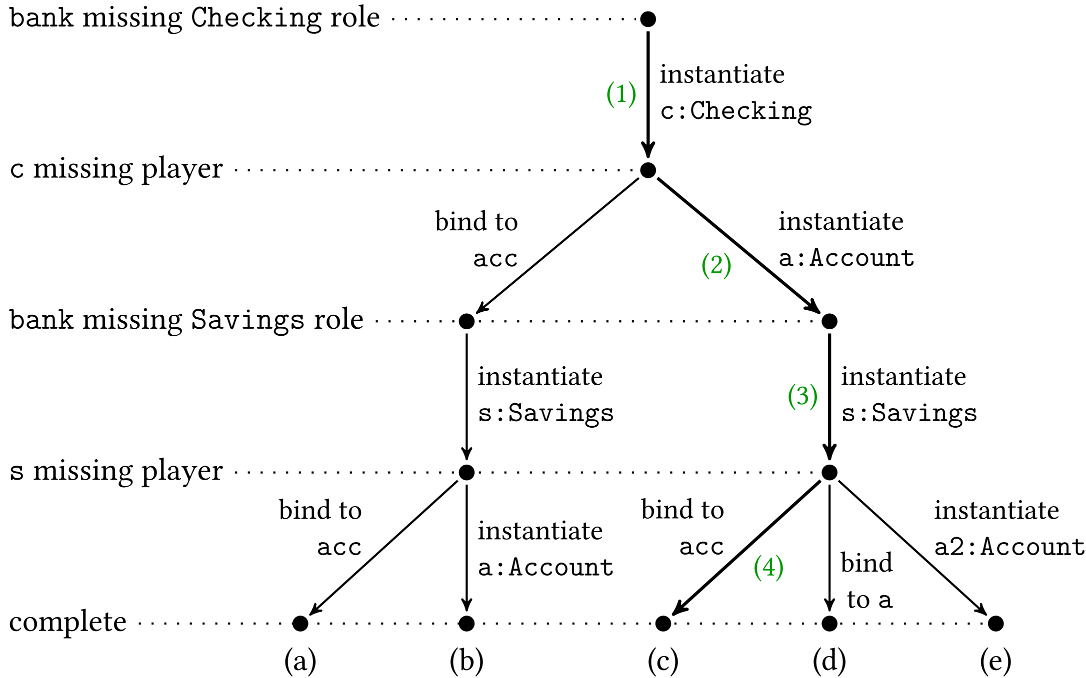


Figure 6.6: Decision tree for extending the incomplete system definition shown in Listing 6.5

sions at every step is shown in Figure 6.6. In the first step, a role of type `Checking` must be assigned to the `bank` to fulfill the `bank`'s first occurrence constraint. Since there is no existing role of that type, the only possible extension is to instantiate a new `Checking` role `c` and assign it to the `bank`. Of course, this newly created role does not have a player yet, so the next step is to bind the role. At this point, there are two possible choices for extending the constraint set. Either `c` is bound to the existing natural instance `acc`, as it has the right type, or a new `Account` is instantiated as the player. Let us choose the latter possibility and follow the right edge in the decision tree. Now that the `bank` compartment has the `Checking` role `c`, only a role of type `Savings` is missing. Again, the only possible extension is to create a new `Savings` instance, since no such role exists yet. As before, this new role does not have a player and may be bound to the existing `acc` natural or a newly created instance. However, since in a previous step the `Account` natural `a` has been instantiated, binding the role `s` to `a` is also an option. We choose the first extension which leaves us with a complete set of constraints (see Listing 6.7) corresponding to the system shown in Figure 6.8c. The `bank` compartment contains a `Checking` role and a `Savings` role and both of these roles are bound to a player. All systems that are associated with a path in the decision tree and therefore correspond to the constraint set in Listing 6.5 are presented in Figure 6.8. ■

Since the constraint language is based on FOL, constraints may also include `forall` and `exists` quantifiers over the set of component instances. A quantifier may optionally spec-

```
1  system {
2    acc : Account;
3    bank : Bank;
4
5    // step (1)
6    c : Checking; c in bank;
7    // step (2)
8    a : Account; c boundto a;
9    // step (3)
10   s : Savings; s in bank;
11   // step (4)
12   s boundto acc;
13 }
```

Listing 6.7: Completed set of constraints for the system in Figure 6.8c

ify a component type or a set of component types to restrict the quantifier to the instances of these types. Furthermore, the built-in type-sets `natural`, `role` and `compartment` may be used to only reason about all instances of naturals, roles and compartments, respectively. Quantification is often necessary in conjunction with incomplete constraint sets which correspond to multiple systems. Since the extension of an incomplete constraint set may instantiate new components that have no corresponding constant symbol in the `system` block, quantification is the only way to reason about these components. Using quantification, general constraints regarding the system structure can be formulated. This, in turn, allows us to filter out systems with an undesired structure.

Example 6.4 (Quantification)**.** Within two of the systems shown in Figure 6.8, neither the `Checking` role nor the `Savings` role in the `bank` compartment are bound to the `acc` component (systems (d) and (e)). This is correct w.r.t. the set of constraints specified in Listing 6.5, but may not have been intended. To rule out these two instances, the system description can be extended as shown in Listing 6.9.

The additional constraint states that there is some role `r` which is contained in the `bank` compartment and played by `acc`. Note that a constraint like `acc boundto c` or `acc boundto r` would not be possible, since the components `c` and `s` have not been defined within the `system` block and thus have no associated constant symbol.

We additionally might require that an account cannot be both a checking account and a savings account. This can be achieved by the constraint shown in Listing 6.10. In this example, type-restricted quantifiers are used to reason only about specific component instances. ■

It is important to note that the quantifiers appearing in a `system` block cannot be used to instantiate new components, i.e., they only range over existing components that have

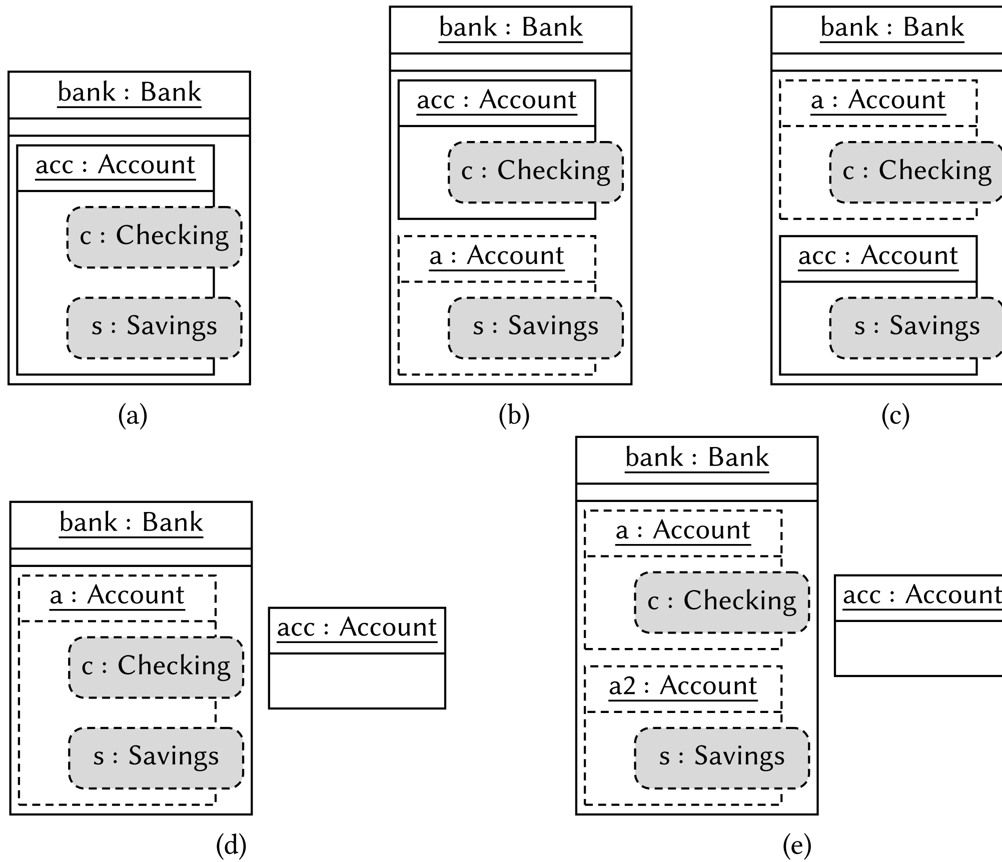


Figure 6.8: Systems corresponding to the constraint set in Listing 6.5. Newly instantiated component instances appear dashed.

```
1 system {
2   acc : Account;
3   bank : Bank;
4
5   exists r : role. r in bank & r boundto acc;
6 }
```

Listing 6.9: Description of systems where `acc` plays a role in the bank

```
1 forall a : Account. !exists c : Checking, s : Savings.
2   c boundto a & s boundto a;
```

Listing 6.10: Constraint stating that an account cannot be a `Checking` and `Savings` account at the same time

been introduced by the “ : ” predicate or the automatic extension of incomplete constraints. This restriction is needed to ensure the finiteness of the systems described by a constraint set.

6.1.2 Definition of operational behavior

The definition of component behavior is strictly separated from the declaration of the system structure. Operational behavior is defined within *modules* that are associated with component types. A module basically represents a single RBA. Upon instantiation of a type, the behavior of a component instance is derived by instantiating the modules associated with its component type. Defining behavior per type enables a generic modeling of component behavior which allows us to easily extend a model by creating new component instances without having to specify behavior for each instance individually.

Modules are associated to a component type by an *implementation declaration* as exemplified in Listing 6.11. The `impl` keyword is followed by the type name for which an implementation is provided, and a list of modules implementing the type. If multiple modules are given, then the behavior of the component type corresponds to the parallel composition of these modules. In case a type is only implemented by a single module that is not used anywhere else, giving the module a name is not necessary. Then, the shorthand syntax as shown in Listing 6.12 can be used, where the `impl` keyword is directly followed by the module body. The additional level of indirection provided by implementation declarations exists mainly for practical reasons. It enables the user of the modeling language to create a library of commonly used modules (e.g., buffers, channels, counters, etc.) which can then be reused in different models and selected by means of an implementation declaration. Furthermore, it allows quickly changing the behavior for certain types by simply swapping out the implementation declarations, without needing to change any module definitions or type declarations.

```
1 natural type Account;
2
3 impl Account(account);
4
5 module account {
6   // ... module body ...
7 }
```

Listing 6.11: An implementation declaration linking a module and a component type

```
1 natural type Account;
2
3 impl Account {
4   // ... module body ...
5 }
```

Listing 6.12: Shorthand notation for providing an implementation for a type

Definition of modules

Modules are defined using an extension of PRISM's language [KNP11] for reactive modules [AH96]. We recall here briefly the structure of a module. A module consists of two parts: a set of local variables that define the state space of the module and a set of guarded commands [JSM97] defining the transitions between those states. All variables have a finite domain and can be either of Boolean or (bounded) integer type. A command describes the transition to another state by updating the local variables of the module, and can only be executed if its guard evaluates to `true`. When the command of a module is executed, it possibly synchronizes with other modules over a shared action. A module in RML has the same basic structure as in the PRISM language, but with some extensions we will discuss in the following.

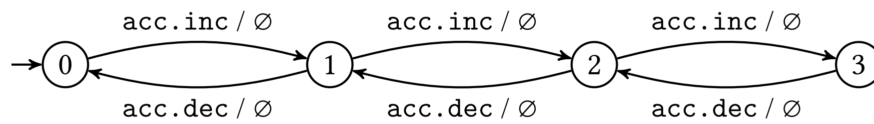
Example 6.5 (Defining the behavior of a natural)**.** Listing 6.13a shows the module describing the behavior of `Account` components. The state space of an account is fully represented by a single variable `balance` which stores how much money is left in the account. Since the model must be finite, we define an upper bound for the account balance (line 10) using a constant (line 1). Increasing and decreasing the balance is defined by the commands in lines 12 and 13, respectively. Figure 6.13b shows the corresponding RBA resulting from the instantiation of the module for the component instance `acc`. ■

While variables in the PRISM language can only have primitive types, variables in RML can also be (possibly multi-dimensional) arrays where the elements are either Boolean or bounded integers. Furthermore, RML has different scoping rules for local variables. In the PRISM language, all variables, including local variables, are defined within the global scope which means that within a model every variable name must be globally unique. In RML, this approach would not be possible. As soon as a module is instantiated more than once, there are multiple copies of the module's variables with the same name[2]. For that reason, a module in RML has its own local (lexical) scope in which the local variables are defined. Outside the module's scope, a reference to a variable of the module must be qualified. The *qualified name* of a variable consists of the component instance name for

[2]The PRISM language provides a rudimentary mechanism for creating a copy of a module, but it requires that all variables of the module must be renamed.

```
1  const MAX_BALANCE = 3;
2
3  natural type Account;
4
5  system {
6    acc : Account;
7  }
8
9  impl Account {
10   balance : [0..MAX_BALANCE] init 0;
11
12   [self.inc] balance < MAX_BALANCE -> (balance' = balance + 1);
13   [self.dec] balance > 0 -> (balance' = balance - 1);
14 }
```

(a) Module defining the behavior of `Account` components



(b) RBA for component instance `acc`

Figure 6.13: A module implementing a component and the corresponding RBA.

```
1 impl Account {
2   balance : [0..MAX_BALANCE] init 0;
3
4   [self.inc[1]] balance <= MAX_BALANCE - 1 -> (balance' = balance + 1);
5   [self.inc[2]] balance <= MAX_BALANCE - 2 -> (balance' = balance + 2);
6   [self.dec[1]] balance >= 1 -> (balance' = balance - 1);
7   [self.dec[2]] balance >= 2 -> (balance' = balance - 2);
8 }
```

Listing 6.14: Indexed action labels for increasing and decreasing the account balance by different amounts

which the module defining the variable has been instantiated, followed by the variable name itself. Consider again the example in Listing 6.13a. Within the body of the module, i.e., in its local scope, the variable `balance` can be used unqualified. However, outside the module body, i.e., in the global scope, the unqualified name `balance` is not defined. To refer to the local variable of the `Account` instance `acc`, the qualified name `acc.balance` must be used. The described scoping rules prevent name clashes in the global scope even in case a module is instantiated multiple times.

Another language construct necessitated by RML's component instantiation mechanism concerns action labels. Suppose a module contains a command labeled with action `act` and is instantiated multiple times. As in the PRISM language, action labels are global. Therefore, all instances of the module would synchronize over the action `act`. This might be the intended behavior in some situations, e.g., if the instances should act in lock-step. However, sometimes the opposite is needed, i.e., that each instance has its own unique action. In the banking scenario, for example, each account should have its own actions for incrementing and decrementing the balance, otherwise all account balances would be linked together. To accommodate the second case, action labels can be prefixed with a component instance name. In Listing 6.13a, the actions are prefixed with the `self` keyword which refers to the component instance name for which the module is instantiated[3]. Therefore, the `Account` instance `acc` provides the actions `acc.inc` and `acc.dec`, as shown in Figure 6.13b. In addition to prefixes, it is also possible to add one or more indices to an action label. For instance, we could amend the account behavior such that the balance can be increased and decreased by different amounts, as shown in Listing 6.14. Indexing action labels is especially useful in conjunction with the metaprogramming constructs of RML (more details are provided at the end of this Section).

[3]The `self` keyword can also be used outside of action labels. For instance, in the module for `Account`, the expression `self.balance` refers to the local variable `balance` of the module.

Role-specific behavior

When defining the operational behavior of roles, additional language constructs can be used within modules to describe role-specific behavior. In particular, adding the `override` keyword in front of an action label marks all corresponding transitions of the command as overriding. That is, for a role instance r the transitions are annotated with $\{+r\}$. Additionally, within a role module, the keyword `player` refers to the component instance to which the role has been bound[4]. Similar to the `self` keyword, `player` may also be used as prefix in action labels to refer to actions prefixed with the player's component name. Note that all transitions originating from a role module are automatically annotated with the respective role names. Therefore, there are no explicit role annotations in RML. Furthermore, each role specified in RML is automatically transformed into a non-blocking role w.r.t. its player (see Section 5.2.1) in order to preserve the original behavior of the player on role-binding.

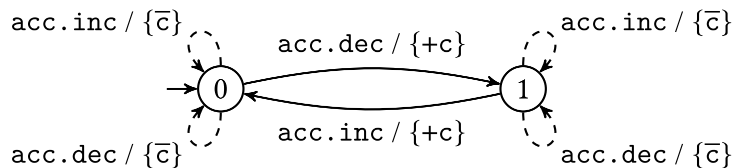
Example 6.6 (Defining role behavior)**.** Figure 6.15a shows the module implementing a `Checking` role. Similar to the `Account` natural (Listing 6.13a), a single variable `od` stores the amount by which the account has been overdrawn (line 12). The role overrides both the `inc` and `dec` actions of its player. Note that since the role instance `c` is bound to the `Account` instance `acc`, the action `player.inc` is resolved to `acc.inc` when instantiating the module for the `Checking` instance `c`. The corresponding RBA for the role instance `c` is presented in Figure 6.15b. The self-loops for the non-blocking behavior of the role are generated automatically and therefore have no corresponding commands in the role module. ■

Some actions of a component are only used for the synchronization between the component's roles and the component itself, i.e., these actions are not intended for the communication with any other component of the system. Therefore, these *internal actions* should only be taken by the player of a role if the role is actually played. In case the role is not played, these actions should be blocked. To achieve this effect, an action within a guarded command can be modified with the `internal` keyword. For instance, we could adapt the module for `Accounts` such that the account balance can only be increased and decreased by the `Source` and `Target` roles. For that, the `inc` and `dec` actions are marked as internal, as shown in Listing 6.16. As mentioned before, all roles are automatically transformed into non-blocking roles w.r.t. their player. However, by adding an `internal` modifier to an action `act` of a component `c`, all roles bound to `c` become blocking for `act`. That is, if any of the bound roles is not played, the action `act` is blocked.

[4] `player` can also be used as a function, even outside of role modules, to get the player of a given role instance. Within a role module, the expressions `player(self)` and `player` are equivalent.

```
1  const MAX_OVERDRAFT = 1;
2
3  role type Checking(Account);
4
5  system {
6    acc : Account;
7    c : Checking;
8    c boundto acc;
9  }
10
11 impl Checking {
12   od : [0..MAX_OVERDRAFT] init 0;
13
14   [override player.dec] od < MAX_OVERDRAFT -> (od' = od + 1);
15   [override player.inc] od > 0 -> (od' = od - 1);
16 }
```

(a) Module defining the behavior of `Checking` role components



(b) RBA for role component instance `c` bound to the component `acc` (auto-generated self-loops for non-blocking behavior appear dashed)

Figure 6.15: A module implementing a role component and the corresponding RBA

```
1 impl Account {
2   balance : [0..MAX_BALANCE] init 0;
3
4   [internal self.inc] balance < MAX_BALANCE -> (balance' = balance + 1);
5   [internal self.dec] balance > 0 -> (balance' = balance - 1);
6 }
```

Listing 6.16: The `Account` module with internal actions

Coordination

The definition of a role-playing coordinator has the same basic structure as a module and is contained in a `coordinator` block. There may be multiple coordinators in a single system which are then combined using parallel composition. Coordinators are instantiated automatically for each system described by the system definition and therefore do not appear in the `system` block. Within a coordinator definition, role-playing is coordinated using an extended form of guarded commands. These *coordination commands* have the general form `[action] [role-guard] guard -> update`. The additional *role guard* is a Boolean expression over the role instances in the system. A coordination command synchronizes with exactly those transitions of the system whose role annotation satisfies the role guard[5]. There is another difference between coordination commands and standard commands. If two coordination commands are independent, i.e., they do not update the same local variables, they can be executed synchronously within a single transition. Since the coordination commands can be seen as rules for role-playing, it makes sense that multiple independent rules can be applied simultaneously. Furthermore, allowing multiple commands to be executed synchronously enables an important optimization which we will discuss in detail in Section 6.2.4.

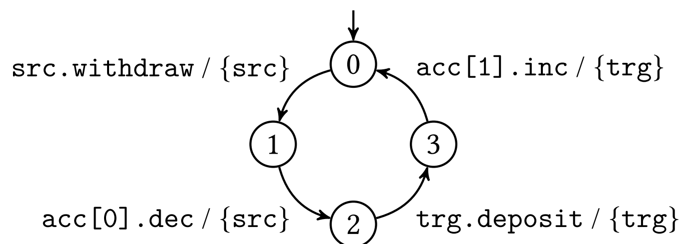
Example 6.7 (Definition of a coordinator)**.** The coordinator in Figure 6.17a controls the money transfer between accounts which play the `src` (source) and `trg` (target) roles. First, it synchronizes with the `src` role over the `withdraw` action to initiate the transfer (line 10). The role guard in this command forces the `src` role to be played. Next, the coordinator allows the `src` role to decrease the account balance (line 11). Note that in accordance with the system description (line 3), the action `player(src).dec` is resolved to `acc[0].dec`. Finally, the money is deposited on the account playing the `trg` role by increasing its balance (lines 12–13). The resulting coordinator RBA is presented in Figure 6.17b. ■

A coordination command can synchronize with exactly those transitions of the system whose role annotation contains at least the roles appearing in the role guard of the

[5] Since the role guard does not concern states but rather concerns transitions, it is surrounded by brackets like the action label.

```
1  system {
2    acc[2] : Account;
3    src : Source; src boundto acc[0];
4    trg : Target; trg boundto acc[1];
5  }
6
7  coordinator {
8    loc : [0..3] init 0;
9
10   [src.withdraw]    [src] loc = 0 -> (loc' = 1);
11   [player(src).dec] [src] loc = 1 -> (loc' = 2);
12   [trg.deposit]     [trg] loc = 2 -> (loc' = 3);
13   [player(trg).inc] [trg] loc = 3 -> (loc' = 3);
14 }
```

(a) Coordinator module for money transfers



(b) Coordinator for the concrete system in (a)

Figure 6.17: Implementation of a coordinator for money transfers and the corresponding automaton

command. In case it is necessary that a coordination command synchronizes over more roles than those appearing in the role guard, an explicit set of roles can be added to the role guard. For instance, the role guard in line 10 of Figure 6.17a can be modified such that it explicitly synchronizes with role-playings containing both the `src` and `trg` roles.

```
[src.withdraw] [src over [src, trg]] l = 0 -> (l' = 1);
```

The role guard in the above example is equivalent to `src & (trg | !trg)`. Additionally, it is possible to specify an explicit set of roles for a whole coordinator.

```
1 coordinator over [src, trg] {
2   // ... coordinator body ...
3 }
```

Besides specifying rules for role-playing, the coordinator may also resolve the nondeterminism of role-playing that arises if two or more roles may be played for some action. In programming languages, the concept of selecting the concrete implementation or functionality for an operation is known as *dispatch* [Boc+12]. Within role-based systems, it is often natural to prioritize the more specific role or the role that has been bound last. RML provides the built-in function `playable` to simplify specifying priorities. For some role instance `r`, the expression `playable(r)` evaluates to `true` if `r` can be played in the current system state, i.e., the current state has an outgoing transition whose role annotation contains `r`. Listing 6.18 exemplifies the use of the `playable` function for defining priorities. Here, the role `b` may only be played if the role `a` cannot be played, effectively giving `a` priority over `b`. Since the guard of a coordination command can be arbitrarily complex, a more sophisticated dispatch is possible as well. This allows us to model dynamic dispatch as it is provided by the role-based language SCROLL [LA15] where the role-playing may depend on the system state.

```
1 coordinator {
2   [] [ a & !b]  playable(a) -> true;
3   [] [!a &  b] !playable(a) -> true;
4 }
```

Listing 6.18: Using the `playable` function to define priority of role `a` over role `b`

Metaprogramming constructs

RML provides several metaprogramming constructs which enable a generic description of a system's behavior. A model may be parametrized, e.g., to enable scaling of the model via a size parameter. Furthermore, if the `system` block of the model defines a family of systems, the metaprogramming constructs allow us to define a single generic description of the behavior covering all system instances, instead of defining the behavior

```
1 function succ(n : int) : int = n + 1;
2
3 function factorial(n : int) : int =
4   if n <= 1 then 1 else n * factorial(n - 1);
5
6 function core(c : component) : component =
7   if c : role then core(player(c)) else c;
8
9 function boundin(n : component, c : compartment) : bool =
10   exists r : role. r boundto n & r in c;
```

Listing 6.19: Function definitions in RML

of each system instance separately. The metalanguage is fully integrated into RML which has two main advantages. First, the generated expressions and commands are always syntactically valid and well-typed. When using an external metaprogramming or template language, these properties are usually not guaranteed. Second, within the metaprogramming constructs, complete access to the model's constants is provided. This includes the component instances defined in the `system` block and those generated by the automatic completion of the constraint set. Paired with the ability to define array variables and indexing action labels, the metaprogramming constructs allow a succinct and generic modeling of system families.

RML supports the definition of functions to reduce code duplication. Functions without parameters are comparable to the `formula` construct in the PRISM language which assigns a shorthand name to an expression. Within a function definition, the function name is followed by a possibly empty list of typed parameters. After the parameter list, the return type of the function is specified. Finally, the function body is an expression (whose type must match the specified return type) which may refer to any parameter specified in the parameter list. Note that this expression may also call any function which allows the definition of recursive and mutually recursive functions. In an RML model, functions can be called within any expression, including constraints in the `system` block.

Example 6.8 (Function definitions)**.** Listing 6.19 shows several illustrative function definitions.

`succ`: This function simply adds 1 to its argument and returns the result. Note that the parameter type `int` does not need to be bounded since the parameter is not part of the model's state.

`factorial`: The second definitions illustrates that the function being defined can be called inside the function's body, enabling recursive definitions.

`core`: In the literature, the player of a role is called the *core object* by some authors. In RML, the component referred to by the `player` keyword and the actual player of

```
system {
  trans[NUM_TRANSACTIONS] : Transaction;
  // ...
}

coordinator {
  loc : array NUM_TRANSACTIONS of [0..3] init 0;

  forall i : [0 .. NUM_TRANSACTIONS - 1] {
    forall src : Source. src in trans[i] {
      forall trg : Target. trg in trans[i] {
        [src.withdraw]   [src] loc[i] = 0 -> (loc[i]' = 1);
        [core(src).dec]  [src] loc[i] = 1 -> (loc[i]' = 2);
        [trg.deposit]    [trg] loc[i] = 2 -> (loc[i]' = 3);
        [core(trg).inc]  [trg] loc[i] = 3 -> (loc[i]' = 3);
  } } }
}
```

Listing 6.20: Coordination of all `Transaction` compartments using `forall`

the role are not necessarily the same. Consider again the example in Listing 6.4. Here, the `Source` role `src` is not directly bound to the `Account` instance `acc`, but rather to the `Checking` role `ch`. Therefore, the expression `player(src)` evaluates to `ch`. However, logically the player of `src` is the account `acc`. The `core` function repeatedly applies the `player` function until the core component of the given role is found. The expression `core(src)` therefore evaluates to `acc`.

`boundin`: In the example in Listing 6.9, we defined a constraint stating that the component `acc` should be bound to some role in the `bank` compartment. It is possible to abstract this constraint into a user-defined predicate `boundin`. Then, the system description can be written as follows[6].

```
system {
  acc : Account; bank : Bank;
  acc boundin bank;
}
```

■

Besides its use in the specification of constraints, the `forall` quantifier can also be employed within modules to generate sequences of commands, stochastic updates and variable assignments. This is especially useful if the behavior of a component depends on the system's structure or size.

[6]Functions with exactly two parameters may be written in infix notation between their arguments.

```
1 impl Account {
2   balance : [0..MAX_BALANCE] init 0;
3
4   forall i : [1..2] {
5     [self.inc[i]] balance <= MAX_BALANCE - i -> (balance'= balance + i);
6     [self.dec[i]] balance >= i -> (balance' = balance - i);
7   }
8 }
```

Listing 6.21: Applying `forall` in conjunction with indexed action labels

Example 6.9 (Coordinator for all `Transaction` compartments)**.** Listing 6.20 shows how the `forall` quantifier can be applied to coordinate multiple `Transaction` compartments using a single generic coordinator. For the definition of the coordinator, we assume that the `system` block defines an array of `Transaction` compartments (line 2). The coordinator keeps the state of each transaction separately in the `loc` array (line 7). The outer `forall` iterates over the indices of the `trans` array elements (line 9). The next `forall`-block iterates over all component instances of type `Source` that are contained in the `Transaction` compartment with index `i` (line 10). An analogous block is used to iterate over all `Target` roles contained in the compartment (line 11). Finally, using the iteration variables defined before, the coordination commands are specified (lines 12–15). Defining the coordinator in this generic form allows us to change the number of transactions in the system (by changing the `NUM_TRANSACTIONS` parameter) or the assignment of the `Source` and `Target` roles to the `Transaction` compartments freely, without having to adapt the coordinator. ■

Example 6.10 (Indexed action labels and `forall`)**.** We revisit here the example presented in Listing 6.14 where the `inc` and `dec` actions are indexed with the amount that is added and subtracted from the account balance, respectively. Using the `forall` construct, we can now write this module more succinctly, as shown in Listing 6.21. Note that the iteration variable `i` is used as the action-label index to generate separate actions for each amount. ■

Costs and rewards

In order to reason about quantitative properties, the states and transitions of a role-based model can be augmented with costs and rewards, such as energy consumption and throughput. As in the PRISM language, transition rewards are associated with an action label. Additionally, rewards can be attached to role-playing by specifying a role guard like in coordination commands. Furthermore, the metaprogramming constructs described in the previous section are also applicable within reward structures.

```
1 rewards "fees" {
2   forall src : Source {
3     [src.withdraw] [src] true := 0.05;
4   }
5 }
```

Listing 6.22: Definition of rewards

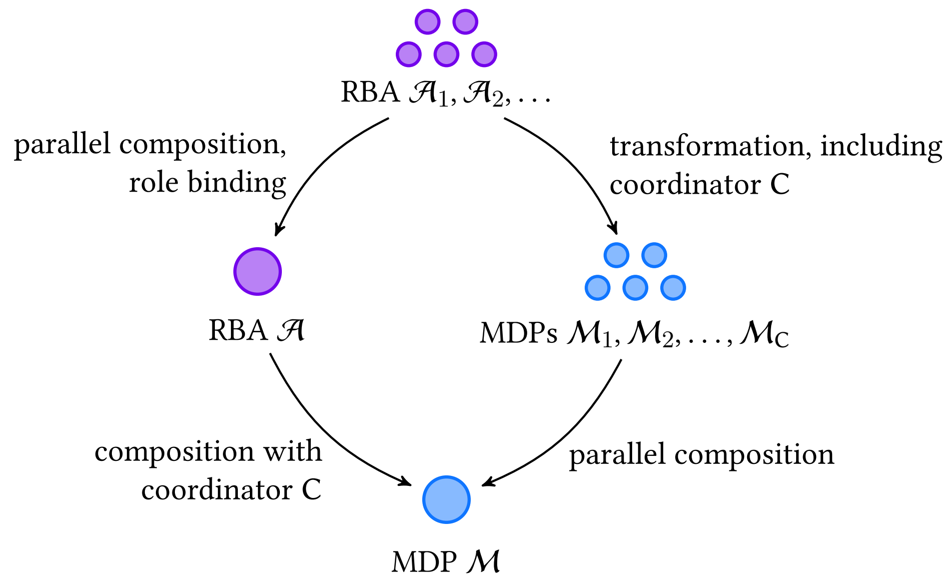


Figure 6.23: Approaches for the composition of RBA

Example 6.11 (Definition of rewards)**.** In Listing 6.22, a reward structure for transaction fees is defined. Using a `forall`-block, a reward item for each `Source` role in the model is generated. The reward item associates a cost of 0.05 with the `withdraw` action. The role guard states that the cost applies if the `src` role is played on this transition. ■

6.2 Translation of role-based models

We employ a translational approach for the analysis of role-based models given in the modeling language RML. First, an RML model is translated into the input language of the model checker PRISM. Then, the analysis is performed by invoking PRISM on the translated model. In case the RML model describes a family of systems, a separate PRISM model for each family instance is generated.

The MDP semantics of RBA (see Section 5.2.3) enables a straightforward translation approach. In a first step, the RBA for the individual components are combined by role-binding and parallel composition which yields a single RBA representing the whole

role-based system. Then, the composition with the role-playing coordinator results in a single MDP, as shown on the left side of Figure 6.23. The aforementioned composition operators can be lifted to act directly on the guarded commands of RML modules. The translation then yields a single PRISM module representing the behavior of the role-based system.

The straightforward translation approach, however, has limited scalability due to technical limitations. The main issue is the possible exponential blow-up of the number of guarded commands caused by the composition of the RML modules. Even for medium-sized models, the translated PRISM model becomes so large that parsing, semantic checking, and model construction dominate the overall analysis time. For larger models, parsing may not even be possible due to memory limits.

There is an alternative translation approach that avoids the exponential blow-up of the model representation. It is possible to transform an RBA into a *Markov decision process with multi-actions* (maMDP) such that the effects of role-binding are mimicked by the parallel composition of maMDPs. In an maMDP, transitions are labeled with sets of actions instead of single actions. The main idea of the transformation is to encode the role-playing annotations into the multi-actions. When transforming an RBA into an maMDP, the number of transitions only grows linearly. The behavior of the whole role-based system then arises from the parallel composition of the transformed maMDPs, as shown on the right side of Figure 6.23. Note that both translation approaches yield isomorphic MDPs. The RBA-to-maMDP transformation can also be lifted to guarded commands. Thus, an RML module can be transformed into a PRISM module with multi-actions. In [Bai+18], we presented an extended version of PRISM that is capable of processing and analyzing models with multi-actions. With this, the second step of the approach, the parallel composition of the maMDPs, can be performed by PRISM during model construction. Therefore, there is no exponential blow-up of the model representation.

In the following, the alternative translation approach outlined above is described in more detail. We start with a formal definition of the RBA transformation in Section 6.2.1 and prove its correctness. Next, an overview of the multi-action extension of PRISM is provided in Section 6.2.2 which yields the foundation for the transformation of RML modules to PRISM modules in Section 6.2.3. Finally, the translation of the role-playing coordinator is discussed in Section 6.2.4.

6.2.1 Transformation to multi-action MDPs

In this section, we formally define the transformation of an RBA to an maMDP. To preserve the effects of role-binding upon parallel composition of these maMDPs, we further define a closure operation on maMDPs modifying the maMDP of the role player depending on the bound role. By applying both the transformation and the closure operation, the effects of parallel composition and role-binding on RBA can be captured using the parallel composition of the transformed maMDPs.

Multi-action MDPs

MaMDPs are a natural extension of MDPs where transitions are labeled with action sets instead of single actions.

Definition 6.12 (maMDP)**.** A *multi-action MDP* is a tuple $\mathcal{M} = (S, Act, \longrightarrow, S^{init})$ where

- S is a finite set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times \mathcal{P}(Act) \times Distr(S)$ is a transition relation with multi-actions, and
- $S^{init} \subseteq S$ is a set of initial states.

The different transition relation compared to MDPs requires us to adapt the parallel composition operator for maMDPs. The parallel composition of maMDPs defined in the following is derived from the composition of a data-abstract variant of *constraint automata* (CA) [Bai+06], or more precisely *simple probabilistic constraint automata* (SPCA) [Bai05].

Definition 6.13 (Parallel composition of maMDP)**.** The *parallel composition* of two multi-action MDPs $\mathcal{M}_i = (S_i, Act_i, \longrightarrow_i, S_i^{init})$ with $i \in \{1, 2\}$ for a set of non-synchronizing actions $N \subseteq Act_1 \cup Act_2$ is defined as

$$\mathcal{M}_1 \parallel_N \mathcal{M}_2 = \big(S_1 \times S_2,\ Act_1 \cup Act_2,\ \longrightarrow,\ S_1^{init} \times S_2^{init} \big)$$

where $\longrightarrow$ is the smallest transition relation fulfilling the rules shown in Figure 6.24.

$$(\mathrm{int}_1)\,\frac{s_1 \xrightarrow{\Sigma}_1 \lambda_1 \qquad \Sigma \cap Act_2 = \varnothing}{\langle s_1, s_2 \rangle \xrightarrow{\Sigma} \lambda_1 * Dirac(s_2)} \qquad (\mathrm{int}_2)\,\frac{s_2 \xrightarrow{\Sigma}_2 \lambda_2 \qquad \Sigma \cap Act_1 = \varnothing}{\langle s_1, s_2 \rangle \xrightarrow{\Sigma} Dirac(s_1) * \lambda_2}$$

$$(\mathrm{sync})\,\frac{s_1 \xrightarrow{\Sigma_1}_1 \lambda_1 \qquad s_2 \xrightarrow{\Sigma_2}_2 \lambda_2 \qquad \Sigma_1 \cap Act_2 = \Sigma_2 \cap Act_1 \qquad |(\Sigma_1 \cup \Sigma_2) \cap N| \leq 1}{\langle s_1, s_2 \rangle \xrightarrow{\Sigma_1 \cup \Sigma_2} \lambda_1 * \lambda_2}$$

Figure 6.24: Rules for the parallel composition of maMDPs

The (sync) rule formalizes the synchronization of maMDPs via handshaking over common actions in the action alphabets. Note that there are no restrictions regarding the emptiness of $\Sigma_1 \cup Act_2$ and $\Sigma_2 \cup Act_1$. Therefore, transitions that are not labeled with any action contained in the action alphabet of the other maMDP and that are thus "unrelated" can be taken simultaneously. By rules (int_1) and (int_2), these transitions can also be executed without synchronization. Furthermore, transitions labeled with the empty action set can synchronize with any transition.

A slight extension in Definition 6.13 compared to the composition of data-abstract SPCA is the inclusion of the set of non-synchronizing actions N. Actions $\alpha, \beta \in N$ can

never synchronize even if they are unrelated, i.e., if $\alpha \neq \beta$. This is formalized by the last precondition of rule (sync). In the product maMDP, the action set of each transition contains at most one non-synchronizing action. The concept of non-synchronizing actions is later needed to distinguish between role-playing actions and "standard" actions of RBA of which the former are synchronizing and the latter are non-synchronizing. Note that the standard parallel composition operator as defined for data-abstract SPCA is obtained by assuming $N = \emptyset$.

Transformation of RBA to multi-action MDPs

We now turn to the transformation of RBA to maMDPs. The main idea here is to treat the role annotations as actions and add them to the transitions' action sets in the maMDP. This transformation makes use of the fact that the action sets of maMDP-transitions are combined upon synchronization (see rule (sync) in Figure 6.24), similar to the combination or role annotations in the parallel composition of RBA (cf. rule (sync) in Figure 5.5).

Before turning to the transformation, we need some auxiliary definitions. Given a set X of role annotations, we define $X^- = \{r : +r \in X\}$. Further, we define the set of overriding actions of a role instance r in an RBA $\mathcal{R} = (S, Act, R, \longrightarrow, S^{init})$ as $Act_{\mathcal{R}}^{+r} = \{\alpha : (s, \alpha, X, \lambda) \in \longrightarrow, +r \in X\}$ where $r \in R$.

Definition 6.14 (maMDP of an RBA)**.** The *maMDP arising from an RBA* $\mathcal{A} = (S, Act, R, \longrightarrow, S^{init})$ is defined as

$$\mathcal{M}[\mathcal{A}] = \big(S,\ Act \cup \mathbb{A}(R),\ \longrightarrow_{\mathcal{M}},\ S^{init}\big)$$

where $\longrightarrow_{\mathcal{M}}$ is the smallest transition relation fulfilling the following rule.

$$\frac{s \xrightarrow{\alpha/X} \lambda}{s \xrightarrow{\{\alpha\} \cup X \cup X^-}_{\mathcal{M}} \lambda}$$

The above transformation is sufficient to imitate the parallel composition of RBA using the parallel composition of the transformed maMDPs, but it does not cover the effects of role-binding. For that, we define an additional *closure operation on maMDPs.* Note that the rules for role-binding (see Figure 5.6) are almost identical to the rules for parallel composition (cf. Figure 5.5). Thus, only a special treatment of transitions with $+r$ annotations is necessary, where r is a role instance name. A role annotation $+r$ means that the role transition will not synchronize with any transition of the player upon binding, i.e., the role takes the transition alone (if it is played) while the player remains in the same state. Suppose the role transition is labeled with some action $\alpha \in Act_r$, where Act_r is the action alphabet of the role's RBA. The case where $\alpha \notin Act_p$, with Act_p being the action alphabet of the player RBA, is already covered by the interleaving rules of the maMDP composition. It remains to handle the case where $\alpha \in Act_p$. Since the action α is shared by

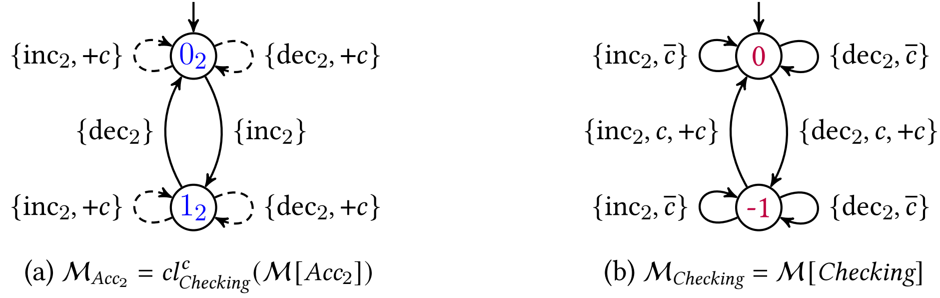


Figure 6.25: maMDPs for the account Acc_2 and the *Checking* role. Transitions added by the closure operation appear dashed.

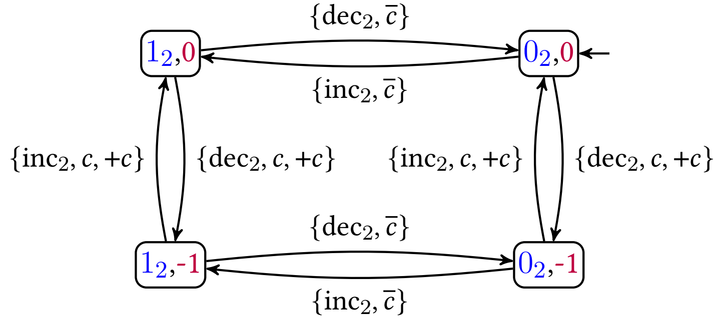


Figure 6.26: maMDP for the composition $cl^{c}_{Checking}(\mathcal{M}[Acc_2]) \parallel_{\{\text{inc}_2,\text{dec}_2\}} \mathcal{M}[Checking]$

the role RBA and the player RBA, their corresponding maMDPs will synchronize over α. However, we must ensure that the player remains in the same state during this transition. In order to achieve this effect, a self-loop labeled with $\{\alpha, +r\}$ is added to every state of the player's maMDP. Note that no other transition of the player's maMDP is labeled with some action set Σ where $+r \in \Sigma$. Therefore, the player will not change its state if the $+r$ transition of the role is taken. In summary, to mimic the effects of role-binding using the parallel composition of maMDPs, it suffices to extend the maMDP of the player depending on the overriding transitions of the role. This extension is formalized by the following definition.

Definition 6.15 (RBA closure)**.** Given an RBA $\mathcal{R}$ and an instance name r, the *closure* over an maMDPs $\mathcal{M} = (S, Act, \longrightarrow, S^{init})$ is defined as

$$cl^{r}_{\mathcal{R}}(\mathcal{M}) = \big(S,\ Act \cup \{+r\},\ \longrightarrow',\ S^{init}\big)$$

where $\longrightarrow' = \longrightarrow \cup \ \big\{\big(s, \{\alpha, +r\}, Dirac(s)\big) \ : s \in S, \alpha \in Act^{+r}_{\mathcal{R}}\big\}$.

Example 6.16 (Transformation and closure)**.** Figure 6.25 shows the maMDPs arising from the respective RBA of the Acc_2 natural and the *Checking* role (cf. Figures 5.3 and 5.4b).

Since the *Checking* role will be bound to the account Acc_2, we further apply the closure operation to the maMDP of Acc_2. The result of the parallel composition of the two maMDPs is presented in Figure 6.26. Note that the resulting maMDP is indeed isomorphic to the RBA $Checking[c \rightarrow Acc_2]$ (cf. Figure 5.7b) when ignoring the additional $+c$ actions.

The example illustrates how the transformation preserves the effect of the $+c$ role annotations. For instance, the transition in $\mathcal{M}_{Checking}$ labeled with $\{\text{inc}_2, c, +c\}$ can only synchronize with transitions in $\mathcal{M}_{Acc_2}$ that are labeled with $\{\text{inc}_2, +c\}$. Since only the self-loops in $\mathcal{M}_{Acc_2}$ carry this action set, the player remains in the original state if the role is played. Also note that the actions $\{\text{inc}_2\}$ of $\mathcal{M}_{Acc_2}$ and $\{\text{inc}_2, c, +c\}$ of $\mathcal{M}_{Checking}$ cannot synchronize, since $\{\text{inc}_2\} \cap Act_{\mathcal{M}_{Checking}} = \{\text{inc}_2\}$ and $\{\text{inc}_2, c, +c\} \cap Act_{\mathcal{M}_{Acc_2}} = \{\text{inc}_2, +c\}$. ■

The compatibility of role-binding and the parallel composition of RBA with the parallel composition of maMDPs under the transformations defined in Definitions 6.14 and 6.15 is given by the following theorem.

Theorem 6.17 (Composition compatibility). *Let $\mathcal{A}_1, \mathcal{A}_2$ be RBA with compatible role interfaces, Act_1, Act_2 be the action sets of $\mathcal{A}_1, \mathcal{A}_2$ respectively, $\mathcal{R}$ be an RBA for role r that can be bound to $\mathcal{A}_1$ and $\cong$ stand for isomorphism (equality up to renaming of states). Then*

1. $\mathcal{M}\big[\mathcal{A}_1 \parallel \mathcal{A}_2\big] \;\cong\; \mathcal{M}[\mathcal{A}_1] \parallel_N \mathcal{M}[\mathcal{A}_2]$
2. $\mathcal{M}\big[\mathcal{R}[r \rightarrow \mathcal{A}_1]\big] \;\cong\; cl^r_{\mathcal{R}}(\mathcal{M}[\mathcal{A}_1]) \parallel_N \mathcal{M}[\mathcal{R}]$

where $N = Act_1 \cup Act_2$.

Proof. We first show (1). Let $\mathcal{A}_i = (S_i, Act_i, R_i, \longrightarrow_i, S^{init}_i)$ for $i \in \{1, 2\}$. Further, let $\mathcal{M}_j = (S_{\mathcal{M}_j}, Act_{\mathcal{M}_j}, \longrightarrow_{\mathcal{M}_j}, S^{init}_{\mathcal{M}_j})$ for $j \in \{1, 2, 21, 22\}$ be maMDPs with $\mathcal{M}_1 = \mathcal{M}\big[\mathcal{A}_1 \parallel \mathcal{A}_2\big]$, $\mathcal{M}_{21} = \mathcal{M}[\mathcal{A}_1]$, $\mathcal{M}_{22} = \mathcal{M}[\mathcal{A}_2]$ and $\mathcal{M}_2 = \mathcal{M}_{21} \parallel_N \mathcal{M}_{22}$. We show that $\mathcal{M}_1 \cong \mathcal{M}_2$. $S_{\mathcal{M}_1} = S_{\mathcal{M}_2}$, $Act_{\mathcal{M}_1} = Act_{\mathcal{M}_2}$ and $S^{init}_{\mathcal{M}_1} = S^{init}_{\mathcal{M}_2}$ follows directly from Definitions 5.4, 6.13 and 6.14. It remains to show that $\longrightarrow_{\mathcal{M}_1} = \longrightarrow_{\mathcal{M}_2}$.

(⊆): The transitions of $\mathcal{M}_1$ arise from the parallel composition of the two RBA $\mathcal{A}_1, \mathcal{A}_2$ defined by the rules in Figure 5.5 which we consider separately by case distinction.

(int$_1$) Assume there is a transition $s_1 \xrightarrow{\alpha_1/X}_1 \lambda_1$ with $\alpha_1 \in Act_1 \setminus Act_2$. Then, by applying rule (int$_1$) of Figure 5.5 and subsequently applying Definition 6.14, it follows that for all $s_2 \in S_2$ the maMDP $\mathcal{M}_1$ has transitions

$$\langle s_1, s_2 \rangle \xrightarrow{\{\alpha_1\} \cup X \cup X^-}_{\mathcal{M}_1} \lambda_1 * Dirac(s_2) \;\;.$$

On the right hand side, the transformation of Definition 6.14 applied to $\mathcal{A}_1$ yields the transition $s_1 \xrightarrow{\{\alpha_1\} \cup X \cup X^-}_{\mathcal{M}_{21}} \lambda_1$. We apply rule (int$_1$) in Figure 6.24, which yields for all $s_2 \in S_2$ the transitions

$$\langle s_1, s_2 \rangle \xrightarrow{\{\alpha_1\} \cup X \cup X^-}_{\mathcal{M}_2} \lambda_1 * Dirac(s_2) \;\;.$$

(int$_2$) The symmetric case follows from case (int$_1$) by swapping $\mathcal{A}_1$ and $\mathcal{A}_2$.

(sync) Assume there are transitions $s_1 \xrightarrow{\alpha_1/X}_1 \lambda_1$ and $s_2 \xrightarrow{\alpha_2/Y}_2 \lambda_2$ with $\alpha = \alpha_1 = \alpha_2$. Then, by applying rule (sync) of Figure 5.5 and subsequently applying Definition 6.14, we obtain that $\mathcal{M}_1$ has a transition

$$\langle s_1, s_2 \rangle \xrightarrow{\{\alpha\} \cup (X \cup Y) \cup (X \cup Y)^-}_{\mathcal{M}_1} \lambda_1 * \lambda_2 \ .$$

On the right hand side, the transformation of Definition 6.14 applied to $\mathcal{A}_1$ and $\mathcal{A}_2$ yields transitions $s_1 \xrightarrow{\{\alpha\} \cup X \cup X^-}_{\mathcal{M}_{21}} \lambda_1$ and $s_2 \xrightarrow{\{\alpha\} \cup Y \cup Y^-}_{\mathcal{M}_{22}} \lambda_2$. Rule (sync) of Figure 6.24 applies, since $|(\{\alpha\} \cup X \cup X^- \cup Y \cup Y^-) \cap N| = 1$ where $\alpha \in N$. Note that $X \cap N = \emptyset$ and $Y \cap N = \emptyset$. This yields the transition

$$\langle s_1, s_2 \rangle \xrightarrow{\{\alpha\} \cup X \cup X^- \cup Y \cup Y^-}_{\mathcal{M}_2} \lambda_1 * \lambda_2 \ ,$$

which is equal to the transition in $\mathcal{M}_1$ since $(X \cup Y)^- = X^- \cup Y^-$.

($\supseteq$): This direction can be shown analogously to ($\subseteq$).

Now let us show (2). Let $\mathcal{A}_p = (S_p, Act_p, R_p, \longrightarrow_p, S_p^{init})$ and $\mathcal{R} = (S_a, Act_a, R_a, \longrightarrow_a, S_a^{init})$ be RBA with $\mathcal{A}_p$ and $\mathcal{R}$ having compatible role interfaces. Further, let $\mathcal{M}_j = (S_{\mathcal{M}_j}, Act_{\mathcal{M}_j}, \longrightarrow_{\mathcal{M}_j}, S_{\mathcal{M}_j}^{init})$ for $j \in \{1, 2, 21, 22\}$ be maMDPs with $\mathcal{M}_1 = \mathcal{M}\big[\mathcal{R}[r \to \mathcal{A}_p]\big]$, $\mathcal{M}_{21} = cl_{\mathcal{R}}^r\big(\mathcal{M}[\mathcal{A}_p]\big)$, $\mathcal{M}_{22} = \mathcal{M}[\mathcal{R}]$ and $\mathcal{M}_2 = \mathcal{M}_{21} \parallel_N \mathcal{M}_{22}$. We show that under a projection π that removes all "+"-prefixed actions, $\mathcal{M}_1 = \pi(\mathcal{M}_2)$. $S_{\mathcal{M}_1} = S_{\mathcal{M}_2}$, $Act_{\mathcal{M}_1} = Act_{\mathcal{M}_2}$ and $S_{\mathcal{M}_1}^{init} = S_{\mathcal{M}_2}^{init}$ follows directly from Definitions 5.5 and 6.13 to 6.15. It remains to show that $\longrightarrow_{\mathcal{M}_1} = \pi(\longrightarrow_{\mathcal{M}_2})$.

($\subseteq$): The transitions of $\mathcal{M}_1$ arise from binding role r in $\mathcal{R}$ to $\mathcal{A}_p$ as defined by the rules in Figure 5.6 which we consider separately by case distinction.

(int$_p$) Assume there is a transition $s_p \xrightarrow{\alpha_p/Y}_p \lambda_p$ with $\alpha_p \in Act_p \setminus Act_a$. Then, by applying rule (int$_p$) of Figure 5.6 and subsequently applying Definition 6.14, it follows that for all $s_a \in S_a$ the maMDP $\mathcal{M}_1$ has transitions

$$\langle s_a, s_p \rangle \xrightarrow{\{\alpha_p\} \cup Y \cup Y^-}_{\mathcal{M}_1} Dirac(s_a) * \lambda_p \ .$$

On the right hand side, the transformation of Definition 6.14 yields the transition $s_a \xrightarrow{\{\alpha_a\} \cup X \cup X^-}_{\mathcal{M}_{21}} \lambda_a$. We apply rule (int$_2$) of Figure 6.24, which for all $s_a \in S_a$ yields the transitions

$$\langle s_a, s_p \rangle \xrightarrow{\{\alpha_p\} \cup Y \cup Y^-}_{\mathcal{M}_2} Dirac(s_a) * \lambda_p \ .$$

(int$_\mathrm{a}$) This case can be shown similarly to (int$_\mathrm{p}$).

(sync) Assume there are transition $s_p \xrightarrow{\alpha_p/X}_p \lambda_p$ and $s_a \xrightarrow{\alpha_a/Y}_{\mathcal{R}} \lambda_a$ with $\alpha = \alpha_p = \alpha_a$ and $+r \notin X$. Then, by applying rule (sync) of Figure 5.6 and subsequently applying Definition 6.14, it follows that $\mathcal{M}_1$ has a transition

$$\langle s_a, s_p \rangle \xrightarrow{\{\alpha\}\cup(X\cup Y)\cup(X\cup Y)^-}_{\mathcal{M}_1} \lambda_a * \lambda_p \ .$$

On the right hand side, the transformation in Definition 6.14 yields the transitions $s_a \xrightarrow{\{\alpha\}\cup X\cup X^-}_{\mathcal{M}_{21}} \lambda_1 * Dirac(s_p)$ and $s_p \xrightarrow{\{\alpha\}\cup Y\cup Y^-}_{\mathcal{M}_{22}} \lambda_p$. Since $(\{\alpha\} \cup X \cup X^-) \cap Act_{\mathcal{M}_{22}} = (\{\alpha\} \cup Y \cup Y^-) \cap Act_{\mathcal{M}_{21}} = \{\alpha\}$ and $|\{\alpha\} \cup X \cup X^- \cup Y \cup Y^-) \cap N| = 1$, rule (sync) in Figure 6.24 applies. Note that $X \cap N = \emptyset$ and $Y \cap N = \emptyset$. This yields the transition

$$\langle s_a, s_p \rangle \xrightarrow{\{\alpha\}\cup X\cup X^-\cup Y\cup Y^-}_{\mathcal{M}_2} \lambda_a * \lambda_p \ .$$

(add) Assume there is a transition $s_a \xrightarrow{\alpha/X}_a \lambda_a$ where $\alpha \in Act_a \cap Act_p$ and $+r \in X$. Then, by applying rule (add) of Figure 5.6 and subsequently applying Definition 6.14, it follows that for all $s_p \in S_p$ the maMDP $\mathcal{M}_1$ has transitions

$$\langle s_a, s_p \rangle \xrightarrow{\{\alpha\}\cup(X\setminus\{+r\})\cup X^-}_{\mathcal{M}_1} \lambda_a * Dirac(s_p) \ .$$

On the right hand side, $\mathcal{M}_{21}$ has transitions $s_p \xrightarrow{\{\alpha,+r\}}_{\mathcal{M}_{21}} Dirac(s_p)$ for all $s_p \in S_p$ introduced by the closure as defined in Definition 6.15. The transformation in Definition 6.14 applied to $\mathcal{R}$ yields that $\mathcal{M}_{22}$ has a transition $s_a \xrightarrow{\{\alpha\}\cup X\cup X^-}_{\mathcal{M}_{22}} \lambda_a$. Applying rule (sync) of Figure 6.24, which is possible since $+r \in X$, yields for all $s_p \in S_p$ the transitions

$$\langle s_a, s_p \rangle \xrightarrow{\{\alpha,+r\}\cup X\cup X^-}_{\mathcal{M}_2} \lambda_a * Dirac(s_p) \ .$$

Removing all $+r$ actions using projection π then yields the transitions

$$\langle s_a, s_p \rangle \xrightarrow{\{\alpha\}\cup(X\setminus\{+r\})\cup X^-}_{\mathcal{M}_2} \lambda_a * Dirac(s_p) \ .$$

(⊇): This direction can be shown analogously to (⊆).

□

Compatibility of RBA and multi-action MDP semantics

After having shown that the composition operators on RBA are compatible with the parallel composition of maMDPs, it is left to show the correctness of the transformation itself (see Definitions 6.14 and 6.15). That is, we need to show that the MDP semantics of an RBA (see Section 5.2.3) corresponds to the maMDP semantics of the transformed RBA. For this, we first define the composition of an maMDP resulting from the transformation of an RBA with an maMDP arising from transforming a role-playing coordinator, i.e., we define the $[\![\cdot]\!]$ operator from Definition 5.11 for maMDPs. To streamline the definition, we introduce the projection functions $act_R(\Sigma) = \Sigma \setminus \mathbb{A}(R)$ and $role_R(\Sigma) = \Sigma \cap \mathbb{A}(R)$.

Definition 6.18 (Composition with role-playing coordinator for maMDPs)**.** The maMDP arising from the composition of an maMDP $\mathcal{M}_a = (S_a, Act_a, \longrightarrow_a, S_a^{init})$ representing a role-based system and an maMDP $\mathcal{M}_c = (S_c, Act_c, \longrightarrow_c, S_c^{init})$ of a role-playing coordinator for a set of roles R is defined as

$$\mathcal{M}_a \bowtie_R \mathcal{M}_c = \left(S_a \times S_c, Act_a \cup Act_c, \longrightarrow, S_a^{init} \times S_c^{init} \right)$$

where $\longrightarrow$ is the smallest transition relation fulfilling the rules in Figure 6.27.

$$(\text{int}_a)\frac{s_a \xrightarrow{\Sigma}_a \lambda_a \qquad act_R(\Sigma) \cap act_R(Act_c) = \varnothing \qquad \mathcal{I} \in \mathbb{R}(role_R(\Sigma), R)}{\langle s_a, s_c \rangle \xrightarrow{act_R(\Sigma) \cup \mathcal{I}} \lambda_a * Dirac(s_c)}$$

$$(\text{int}_c)\frac{s_c \xrightarrow{\Sigma}_c \lambda_c \qquad act_R(\Sigma) \cap act_R(Act_a) = \varnothing \qquad \mathcal{I} \in \mathbb{R}(role_R(\Sigma), R)}{\langle s_a, s_c \rangle \xrightarrow{act_R(\Sigma) \cup \mathcal{I}} Dirac(s_a) * \lambda_c}$$

$$(\text{sync})\frac{\begin{array}{c} s_a \xrightarrow{\Sigma_a}_a \lambda_a \qquad\qquad s_c \xrightarrow{\Sigma_c}_c \lambda_c \\ act_R(\Sigma_a) \cap act_R(Act_c) = act_R(\Sigma_c) \cap act_R(Act_a) \\ |act_R(\Sigma_a \cup \Sigma_c)| = 1 \qquad \mathcal{I} \in \mathbb{R}(role_R(\Sigma_a \cup \Sigma_c), R) \end{array}}{\langle s_a, s_c \rangle \xrightarrow{act_R(\Sigma_a \cup \Sigma_c) \cup \mathcal{I}} \lambda_a * \lambda_c}$$

Figure 6.27: Rules for the composition of an maMDP arising from an RBA and an maMDP arising from a role-playing coordinator

We now show the correctness of the alternative translation approach. Note that the following theorem directly implies behavioral equivalence of an RBA under the trivial coordinator (consisting only of a single state and an empty transition relation) and its corresponding transformed maMDP. First, we define a technical notion of *projections on multi-actions* to relate maMDPs of a certain form with standard MDPs. For a given set of

actions Act and roles R, let $\pi\colon \mathcal{P}(Act \cup \mathbb{A}(R)) \to Act \times \mathcal{P}(R)$ be a function that transforms a multi-action into a pair of action and role-playing where for any $\alpha \in Act_c \cup Act_a$ and $\mathcal{I} \subseteq R$ we have $\pi(\{\alpha\} \cup \mathcal{I}) = \langle \alpha, \mathcal{I} \rangle$. When $\mathcal{M}$ is an maMDP where in each transition at most one element of the action set is not contained in R, we write $\pi(\mathcal{M})$ to denote the MDP arising from the application of π onto the action sets of every transition.

Theorem 6.19 (Correctness of the translation approach). *For any RBA $\mathcal{A}$ and coordinator* C *with roles R_a and R_c, respectively, we have*

$$\pi\big(\, \mathcal{M}[\mathcal{A}] \bowtie_R \mathcal{M}[\mathrm{C}]\,\big) \cong [\![\mathcal{A}]\!]_{\mathrm{C}}$$

where $R = R_a \cup R_c$.

Proof. Let $\mathcal{A} = (S_a, Act_a, R_a, \longrightarrow_a, S_a^{init})$ be an RBA and $\mathrm{C} = (S_c, Act_c, R_c, \longrightarrow_c, S_c^{init})$ be a role-playing coordinator. Further, let $\mathcal{M}_1$ be an maMDP with $\mathcal{M}_1 = \mathcal{M}[\mathcal{A}] \bowtie_R \mathcal{M}[\mathrm{C}]$ and $\mathcal{M}_2$ be an MDP with $\mathcal{M}_2 = [\![\mathcal{A}]\!]_{\mathrm{C}}$, and $R = R_a \cup R_c$. Note that projection π as defined above can be applied to all multi-actions in $\mathcal{M}_1$, since multi-actions in $\mathcal{M}[\mathcal{A}]$ and $\mathcal{M}[\mathrm{C}]$ contain at most one action α with $\alpha \in Act_a$ and $\alpha \in Act_c$, respectively. Furthermore, the (sync) rule of the composition operator $\bowtie_R$ (Figure 6.27) only produces multi-actions which contain at most one action $\alpha \in Act_a \cup Act_c$. We now show that $\pi(\mathcal{M}_1) = \mathcal{M}_2$. First, $S_1 = S_2 = S_a \times S_c$ and $S_1^{init} = S_2^{init}$ is clear by Definitions 6.14 and 6.18. Applying Definitions 6.14 and 6.18 yields $Act_1 = (Act_a \cup \mathbb{A}(R_a) \cup (Act_c \cup \mathbb{A}(R_c)) = Act_a \cup Act_c \cup \mathbb{A}(R)$ and Definition 5.11 yields $Act_2 = (Act_a \cup Act_c) \times \mathcal{P}(R)$, with $R = R_a \cup R_c$. Thus, we have $\pi(Act_1) = Act_2$. Finally, $\pi(\longrightarrow_1) = \longrightarrow_2$ is clear by the definition of $\bowtie_R$ (cf. Definition 6.18) which matches the definition of $[\![\cdot]\!]$ (cf. Definition 5.11). □

6.2.2 Multi-action extension of PRISM

The previously described transformation allows us to translate the individual RBA that constitute a role-based system into maMDPs. Using this foundation, the modules that constitute an RML model can be translated into PRISM modules with multi-actions such that the number of guarded commands in the PRISM model is only polynomial in the number of commands of the corresponding RML model. However, applying the parallel composition for modules with multi-actions directly on the PRISM-language level would again result in an exponential blow-up in the size of the model representation. Thus, to reap the benefits of the alternative translation approach, PRISM must be able to directly handle models containing multi-actions. Then, the composition of the multi-action modules is shifted to the model construction phase, as it is the case for standard PRISM models. The extended version of PRISM capable of processing models with multi-actions is described in this section. The implementation described here has been published in [Bai+18].

A multi-action-capable version of PRISM also has broad applications outside the translation of role-based models. It can, for instance, be applied for modeling exogenous

$$\text{(1a)}\ \frac{[\Sigma]\ g \rightarrow u \in C_1 \qquad \Sigma \cap Act = \emptyset}{[\Sigma]\ g \rightarrow u \in C} \qquad \text{(1b)}\ \frac{[\Sigma]\ g \rightarrow u \in C_2 \qquad \Sigma \cap Act = \emptyset}{[\Sigma]\ g \rightarrow u \in C}$$

$$\text{(1c)}\ \frac{[\Sigma_1]\ g_1 \rightarrow u_1 \in C_1 \qquad [\Sigma_2]\ g_2 \rightarrow u_2 \in C_2 \qquad \Sigma_1 = \Sigma_2 \qquad \Sigma_1 \cap Act \neq \emptyset}{[\Sigma_1 \cup \Sigma_2]\ g_1 \wedge g_2 \rightarrow u_1 * u_2 \in C}$$

$$\text{(2a)}\ \frac{]\Sigma[\ g \rightarrow u \in C_1 \qquad \Sigma \cap Act = \emptyset}{]\Sigma[\ g \rightarrow u \in C} \qquad \text{(2b)}\ \frac{]\Sigma[\ g \rightarrow u \in C_2 \qquad \Sigma \cap Act = \emptyset}{]\Sigma[\ g \rightarrow u \in C}$$

$$\text{(2c)}\ \frac{]\Sigma_1[\ g_1 \rightarrow u_1 \in C_1 \qquad]\Sigma_2[\ g_2 \rightarrow u_2 \in C_2 \qquad \Sigma_1 \cap Act = \Sigma_2 \cap Act}{]\Sigma_1 \cup \Sigma_2[\ g_1 \wedge g_2 \rightarrow u_1 * u_2 \in C}$$

$$\text{(3a)}\ \frac{]\Sigma_1[\ g_1 \rightarrow u_1 \in C_1 \qquad [\Sigma_2]\ g_2 \rightarrow u_2 \in C_2 \qquad \Sigma_2 \neq \emptyset \qquad \Sigma_1 = \Sigma_2 \cap Act}{[\Sigma_1 \cup \Sigma_2]\ g_1 \wedge g_2 \rightarrow u_1 * u_2 \in C}$$

$$\text{(3b)}\ \frac{[\Sigma_1]\ g_1 \rightarrow u_1 \in C_1 \qquad]\Sigma_2[\ g_2 \rightarrow u_2 \in C_2 \qquad \Sigma_1 \neq \emptyset \qquad \Sigma_2 = \Sigma_1 \cap Act}{[\Sigma_1 \cup \Sigma_2]\ g_1 \wedge g_2 \rightarrow u_1 * u_2 \in C}$$

Figure 6.28: Rules for the parallel composition of PRISM modules with multi-actions synchronizing over the action alphabet *Act*

coordination [Bai+18]. Therefore, we chose to conservatively extend the PRISM language in order to maintain compatibility with the standard semantics of PRISM models. A command in the extended PRISM language consists of a multi-action, a state guard, and a stochastic update. The multi-action can take either the *closed* form, denoted $[\Sigma]$, or the *open* form, denoted $]\Sigma[$. Intuitively, a closed multi-action cannot be extended with additional actions during composition. For an open multi-action Σ, composition may yield a multi-action Σ' with $\Sigma \subseteq \Sigma'$. Note that in case the model contains only closed multi-actions consisting of at most one action, it has the same semantics as a standard PRISM model.

We now turn to the parallel composition operator for modules containing multi-actions. Figure 6.28 presents the rules for the $M_1\,|Act|\,M_2$ operator, where M_1, M_2 are modules and *Act* is the synchronization alphabet. The standard parallel composition operator $M_1 \parallel M_2$ is obtained by using $Act = Act_1 \cap Act_2$ as the synchronization alphabet. The expression $[\Sigma]\ g \rightarrow u \in C_i$ denotes that there is a guarded command with closed multi-action Σ, guard g and stochastic update u in module M_i. Analogously, $]\Sigma[\ g \rightarrow u \in C_i$ denotes that there is a command with open multi-action Σ. Furthermore, $u_1 * u_2$ stands for the combined stochastic update of u_1 and u_2 using the product distribution as in the standard PRISM semantics. Rules (1a)–(1c) cover the cases where only closed multi-actions are used. Apart from using multi-actions instead of single actions, these rules are equivalent

to PRISM's standard parallel composition operator. Rules (2a)–(2c) deal with open multi-actions. These rules correspond to the rules for the parallel composition of maMDPs (cf. Figure 6.24) lifted to guarded commands. The last two symmetric rules (3a) and (3b) handle the case where open and closed multi-actions synchronize. Since closed multi-actions cannot be extended with further actions, the result of such a synchronization is again a closed multi-action. Note that from rules (1a)–(1c) it follows that the empty closed multi-action `[]` never synchronizes with any other multi-action. The empty open multi-action `][`, on the other hand, can synchronize with any other open or closed multi-action except the empty closed multi-action. The parallel composition operator for modules with multi-actions is both commutative and associative.

```
1  module M1
2    s : [0..2] init 0;
3
4    ]a, b[ s=0 -> (s'=1);
5    [a, b] s=0 -> (s'=2);
6  endmodule
7
8  module M2
9    t : [0..2] init 0;
10
11   ]b, c[ t=0 -> (t'=1);
12   ]b[    t=0 -> (t'=2);
13 endmodule
```

(a) Individual modules

```
1  module M1_M2
2    s : [0..2] init 0;
3    t : [0..2] init 1;
4
5    ]a, b, c[ s=0 & t=0 -> (s'=1) & (t'=1);
6    ]a, b[    s=0 & t=0 -> (s'=1) & (t'=2);
7    [a, b]    s=0 & t=0 -> (s'=2) & (t'=2);
8  endmodule
```

(b) Composed module

Figure 6.29: Composition of modules with multi-actions

Example 6.20 (Composition of modules with multi-actions)**.** Figure 6.29 shows two PRISM modules with multi-actions and the result of their composition. Since both the commands in line 4 and line 11 of Figure 6.29a feature an open multi-action with shared action `b`, the commands are combined according to rule (2c) in Figure 6.28. Similarly, the command in line 4 can also synchronize with the command in line 12. For the commands in line 5 and line 12, rule (3b) applies. Note that the commands in line 5 and line 11 do not synchronize, since the closed multi-action `[a, b]` cannot be extended with action `c`. ■

Multi-actions can additionally be used within the declarations of reward structures to assign rewards to the composed system. Like in modules, both the open and the closed form of multi-actions may be used. A reward item with a closed multi-action $[\Sigma]$ will assign a reward to transitions with multi-action Σ' if $\Sigma = \Sigma'$. For open multi-actions $]\Sigma[$, the reward is assigned if $\Sigma \subseteq \Sigma'$. The open form is especially useful to assign rewards for some specific action regardless of which other actions are contained in the multi-action as well.

```
natural type Account {
  bal : [0 .. MAX_BALANCE] init 0;

  [self.inc] bal < MAX_BALANCE -> (bal' = bal + 1);
  [self.dec] bal > 0 -> (bal' = bal - 1);
}

role type Checking(Account) {
  od : [0 .. MAX_OVERDRAFT] init 0;

  [override player.dec] od < MAX_OVERDRAFT -> (od' = od + 1);
  [override player.inc] od > 0 -> (od' = od - 1);
}

system {
  a : Account; c : Checking;
  c boundto a;
}
```

Listing 6.30: RML modules defining the behavior of the `Account` natural and `Checking` role

The extensions described previously have been implemented in PRISM's explicit engine and in the (semi-)symbolic engines. The extended PRISM version is able to directly process and analyze the multi-action PRISM models generated by the RBSC tool.

6.2.3 Translation of components

By utilizing the transformation of RBA to maMDPs and the multi-action extension of PRISM described in the previous sections, we are now able to translate RML modules to PRISM modules with multi-actions. Since the translation of the language constructs shared by RML and the PRISM language is straightforward, we focus here only on the role-specific constructs.

Consider again the banking example with a single checking account role bound to an account natural shown in Listing 6.30. First, actions for the role-playing are added to all commands of a role's modules. In the translation of the example in Listing 6.31, the action c is therefore added to the multi-actions of the translated commands (lines 13 and 14). Since the actions within the role module are marked with `override`, the additional action ovr_c (corresponding to the $+c$ role annotation) is added in the translation according to Definition 6.14. All role instances in RML are automatically transformed to be non-blocking w.r.t. their player (see also Section 5.2.1). Therefore, a self-loop for each action of the role must be added to every state of the role. This is achieved by the commands in

```
1  module a
2    a_bal : [0..1] init 0;
3
4    ]a_inc[ a_bal < 1 -> (a_bal' = a_bal + 1);
5    ]a_dec[ a_bal > 0 -> (a_bal' = a_bal - 1);
6    ]a_dec, ovr_c[ true -> true;
7    ]a_inc, ovr_c[ true -> true;
8  endmodule
9
10 module c
11   c_od : [0..1] init 0;
12
13   ]a_dec, c, ovr_c[ c_od < 1 -> (c_od' = c_od + 1);
14   ]a_inc, c, ovr_c[ c_od > 0 -> (c_od' = c_od - 1);
15   ]a_dec, not_c[ true -> true;
16   ]a_inc, not_c[ true -> true;
17 endmodule
```

Listing 6.31: Translated modules for the `Account` natural and `Checking` role. Generated role actions appear in *italic*.

lines 15 and 16 in Listing 6.31 labeled with the *not_c* action (corresponding to the $\bar{c}$ role annotation) indicating that the role is not played on these transitions. For the translation of components that have roles bound to them, the closure operator must be applied (see Definition 6.15). In the example, the checking role `c` is bound to the account `a` (line 17 of Listing 6.30). Thus, the closure is applied to the component implementing `a`, i.e., self-loops for each action of the role `c` labeled with *ovr_c* are added to the module (lines 6 and 7 of Listing 6.31).

According to the definitions of role-binding (Definition 5.5) and the parallel composition of RBA (Definition 5.4), transitions can only synchronize over shared actions which should also be the case for RML modules. However, after translating RML modules to PRISM modules with multi-actions, unrelated actions may synchronize as well, since the translated model contains only open multi-actions. In order to maintain the compatibility of role-binding and the parallel composition of RML modules with the parallel composition of their corresponding translated PRISM modules, the synchronization of unrelated actions must be prevented. Note that the parallel composition of modules with multi-actions within PRISM currently does not support the composition under a set of non-synchronizing actions like in Definition 5.4. Therefore, the following construction is necessary. An additional PRISM module (usually named `Nosync`) is generated as part of the translation. For each action α appearing in the RML model, the `Nosync` module has a self-loop labeled with the open multi-action $]\alpha[$. The generated module for the example

```
module Nosync
  ]a_dec[ true -> true;
  ]a_inc[ true -> true;
endmodule
```

Listing 6.32: Generated module for preventing the synchronization of non-role actions

is shown in Listing 6.32. Since the `Nosync` module covers the whole alphabet of non-role actions and each command is only labeled with a single action, the parallel composition of this module with the translated model effectively removes all transitions arising from the synchronization of unrelated actions. In the example, all transitions labeled with an open multi-action containing both the actions `a_inc` and `a_dec` have no synchronizing transition within the `Nosync` module, and are therefore blocked. Thus, the inclusion of the `Nosync` module mimics the effect of the parallel composition operator for maMDPs defined previously.

6.2.4 Translation of role-playing coordinators

A coordinator may contain both standard commands as well as coordination commands. Since neither can coordinators be roles nor can roles be bound to them, the translation of standard commands is straightforward and merely involves the resolution of qualified identifiers and metaprogramming constructs. The translation of coordination commands, on the other hand, must handle the additional role guard. A role guard is a symbolic representation of a set of role-playings. Therefore, the translation of a coordination command may result in multiple commands, one for each role-playing that satisfies the role guard. A role-playing satisfying the role guard is encoded as a set of actions (analogous to the encoding utilized in the translation of RML modules). These role-playing actions are subsequently added to the multi-action of the translated command.

The number of possible role-playings is exponential in the number of roles. Therefore, the number of generated commands may also grow exponentially when translating a coordinator. This is a major issue even for medium-sized models. However, this issue can be mitigated by using a more optimized translation. The optimization is based on the observation that usually not all roles appear in every role guard. In fact, it is often the case that a coordination command only concerns small groups of roles, e.g., those that are contained in the same compartment, or even just individual roles. A second observation is that for translating a coordinator, the set of possible role-playings can be restricted to the roles that actually appear in some role guard of the coordinator. Combining these two observations leads to the main idea of the optimized translation. The coordinator can be *partitioned* into smaller coordinators, one for each individual group of coordinated roles. Since each of these coordinators handles only a few roles, the set of role-playings

that has to be considered during their translation is smaller than the complete set of all role-playings. While this optimization can be applied by hand when writing the model, its impact on the overall size of the resulting PRISM model is so significant that the implementation performs the partitioning of the coordinator automatically.

In order to apply the partitioning in an automated fashion, we first must establish the conditions under which coordination commands can be separated into different partitions. In particular, two coordination commands with sets of roles R_1, R_2 appearing in the role guards and sets of updated variables V_1, V_2, respectively, must be contained in the same partition if at least one of the following conditions hold.

1. $R_1 \cap R_2 \neq \emptyset$
2. $V_1 \cap V_2 \neq \emptyset$

Condition (1) ensures that commands that coordinate some common set of roles are not put into different coordinators. In the PRISM language, a command can only update a local variable if the command is contained in the same module as the definition of the variable. Thus, if any given two commands update the same variable, they must necessarily be contained in the same module. This is addressed by condition (2). The partitioning of a coordinator proceeds as follows. First, a graph is constructed where the node set is the set of coordination commands of the coordinator. There is an edge between two commands if some of the conditions stated above hold. Then, a new coordinator for each connected component of the graph is generated.

Example 6.21 (Translation of a coordinator)**.** Listing 6.33 contains a coordinator formalizing the rule that the role `a` must have been played before the roles `b` or `c` can be played. This coordinator can be partitioned into two coordinators, resulting in the coordination modules shown in Listing 6.34. Both the first and the second coordination commands update the local variable `played` (lines 19 and 20) and therefore end up in the same module (lines 4 and 5 in Listing 6.34). The third command of the coordinator (line 21) coordinates a different set of roles than the first two and thus can be placed in a different coordinator module (lines 10–12 in Listing 6.34). Note that the translation of the coordinator takes the definitions of the role modules into account. For instance, from the behavioral definition for role type `R` it is deduced that roles `b` and `c` can only be played together for the `tick` action (line 10 in Listing 6.34). The final commands in each of the translated coordinators (lines 6 and 13) are needed for technical reasons. It is necessary that the action alphabet of a coordination module contains all role-playing actions, both in positive and negative form, as well as the actions provided by the coordinated roles. Since the PRISM language does not provide a construct to specify the action alphabet of a module explicitly, a command containing all actions is added to the module. The guard of this command is `false`, therefore it adds no transitions to the resulting MDP. ■

```
1  natural type N;
2
3  role type R(N) {
4    s : [0 .. 2] init 0;
5
6    [self.act] s = 0 -> (s' = 1);
7    [tick] true -> (s' = 2);
8  }
9
10 system {
11   n : N;
12   a : R; b : R; c : R;
13   forall r : R. r boundto n;
14 }
15
16 coordinator {
17   played : bool init false;
18
19   [a.act] [a] !played -> (played' = true);
20   [tick] [a] true -> (played' = true);
21   [] [b | c] played -> true;
22 }
```

Listing 6.33: Coordinator stating that role a must have been played before roles b and c can be played

6.2.5 Encoding role-playing into states

The translation of a role-based model into the PRISM language allows us to utilize PRISM's rich property specification language to define the desired behavior of the system. In order to reason about role-playing which is encoded into the actions of the MDP, we adapt well-known techniques for action-based logics [DV90]. Model checking for action-based temporal logics can be reduced to standard model checking by translating both the formula and the model [FGR94]. The main idea of the model translation is splitting each transition labeled with a visible action and adding a new state labeled with the observed action. An example is shown in Figure 6.35, where the translation adds an intermediate state (α, q) signifying that on the transition from p to q an α-action has occurred. With these intermediate states in place, we can subsequently translate a temporal-logic formula incorporating actions into a formula reasoning purely over states. We will concentrate here only on the model translation and refer to [DV90] for the translation of formulas.

For reasoning about role-playing, the model translation described above has to be adapted. This is necessitated by the fact that for a given observed role r, a transition in an

```
1  module coordinator
2    played : bool init false;
3
4    ]a_act, a[ !played -> (played' = true);
5    ]tick, a[ true -> (played' = true);
6    ]a_act, tick, a, not_a[ false -> true;
7  endmodule
8
9  module coordinator2
10   ]tick, b, c[ played -> true;
11   ]b_act, b[ played -> true;
12   ]c_act, c[ played -> true;
13   ]b, not_b, c, not_c[ false -> true;
14 endmodule
```

Listing 6.34: Translated coordinator, partitioned into two modules. Generated role actions appear in *italic*.

(a) Fragment of a transition system

(b) Translation of the fragment

Figure 6.35: Translation for encoding the visible action α into the state space

RBA may not only be annotated with r (the role is played) and $\overline{r}$ (role is not played), but may also have no r-annotation at all. Remind that this means the role r may be played, but not necessarily. When specifying properties involving role-playing, such transitions need special treatment. Suppose we require that from some point onward, the role r has to be played continuously. Then, a missing annotation for r must be interpreted as "r is played". However, if it is required that r is continuously *not* played after some point, the interpretation must be that "r is not played" on transitions with no r-annotation. The main idea to handle missing role annotations is to introduce three different labellings per role r: role r has been played in the last transition (r), r has explicitly not been played ($\overline{r}$), and thirdly, r may have been played ($r?$). Using these labellings, the previously described examples can be easily expressed. A role r is continuously played on a path if there is no state labelled with $\overline{r}$. Analogously, r is not played continuously if there is no state labelled with r. Similar to the translation for action-based logics, we split transitions annotated with visible role-playing and introduce an intermediate state labelled with the observed role-playing. An example is shown in Figure 6.36. In the first transition, the intermediate state is labelled with s, since role s has been played in the transition, and with $t?$ because the transition carries no t-annotation.

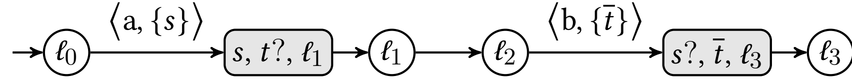


Figure 6.36: Translation for encoding role-playing of roles s and t into the state space

There are two possible ways to implement this translation for encoding the role-playing into the state space. First, the translation could be performed directly on the MDP resulting from the translation of the role-based model. This however would require to adapt the model-construction process in the model checker, in this case PRISM, to insert the additional states. The second possibility is to perform the translation on the PRISM-language level. Since the second approach only requires minor modifications to the translation from an RML model to a PRISM model, the tool applies the translation on the PRISM-language level.

The translation on the language level must accomplish to the following. First, it must add the intermediate states for making the role-playing observable using the state labelling. Second, it must split transitions with role-playing annotations. In order to keep the number of additional states as small as possible, these modifications are not applied for all roles in the model, but only for selected *observed* roles. Adding the intermediate states is achieved by introducing an integer variable with range -1 to 1 for each observed role. In the following we call these variables *role-playing variables*. The possible values for role-playing variables map directly to the labelling described before, with 1, -1, and 0 meaning the role has been played, has not been played, and may have been played, respectively. For each role the role-playing variable, which has the same name as the role, is added to the module specifying the role's behavior. This allows the role module to update its associated role-playing variable. Splitting transitions is achieved as follows. First, an atomic proposition `step` (defined as a `formula` in the PRISM model) is defined which is true if we are in a "normal" state, i.e., not an intermediate state. To make sure that the transitions of the original model are only taken from non-intermediate states, the guard `step` is added to each command of the model (this includes commands of all modules, not limited to role modules). Each command in a role module carrying a role annotation updates the role-playing variable accordingly. Consider the example in Listing 6.37 which shows the translated role module for a role t. The command in line 7 is annotated with t, therefore it writes 1 to the role-playing variable. Similarly, the command in line 8 updates the variable to -1 as it is annotated with not_t. Note that for all transitions in the model that carry neither the t nor the not_t actions, the role-playing variable is not updated and thus keeps the value 0. These modifications cover the first half of the split transitions, into the intermediate states. It remains to add the second half by introducing transitions from the intermediate states to the successor states. For that, there is a reset command in each role module, such as in line 9 of Listing 6.37. Each of these reset commands is labelled with the `reset` action, such that all role modules leave the intermediate state synchronously. The guard is always `!step`, therefore the reset commands can only be

```
1  formula step = s = 0 & t = 0;
2
3  module t
4    t : [-1..1] init 0;
5    t_x : [0..1] init 0;
6
7    ]b, t, ovr_t[ step & t_x = 0 -> (t_x' = 1) & (t' = 1) ;
8    ]b, not_t[ true -> (t' = -1) ;
9    [reset] !step -> (t' = 0);
10 endmodule
```

Listing 6.37: Source-level translation for encoding role-playing into the state space. Additions to the role module are highlighted.

executed in an intermediate state. Finally, all role-playing variables are reset to 0. With these modifications in place, the MDP described by the PRISM model has exactly the desired structure which allows us to observe the role-playing using accordingly labelled intermediate states.

7 Exogenous coordination of roles

The coordination of role-playing is a central aspect in role-based systems. Not only does the coordinator control the adaptations to context changes, but it also encodes the rules for role-playing. This includes both invariants, e.g., that certain roles must not be played at the same time, and also rules specifying the temporal order of role-playing. Since roles only change the behavior of their player when they are actually played, enforcing and prohibiting certain role-playings has a substantial influence on the overall behavior of a role-based system. In the RBA formalism presented in Chapter 5, the coordinator is formalized as an automata-based component similar to all other components of the system. Likewise, in the role-oriented modeling language (see Chapter 6), the coordinator is defined using the same guarded command language that is used for describing the behavior of components. These representations can make the rules for role-playing quite opaque. For instance, the fact that roles a and b must not be played together cannot be captured using a single transition or command. Rather, this rule is enforced by not having any transition or command where a and b appear together in the role-playing annotation. This chapter explores the idea of making the coordination explicit by employing a coordination language for specifying the role-playing within a role-based system.

Coordination languages facilitate a clean separation between coordination logic and application logic. Several coordination languages have been proposed [PA98] which can be classified as either endogenous or exogenous. In *endogenous* coordination, the coordination primitives of the coordination language are used within the implementation of the individual components. Contrarily, in *exogenous* coordination the coordination primitives reside outside the components. Thus, components are orchestrated from "outside" via their interfaces. This allows for a decoupling of component implementations since the individual components do not have to be aware of each other. In the RBA approach, the coordination of role-playing is concentrated in a dedicated coordination component. Thus, it is rather natural to employ an exogenous coordination language. The approach presented in this chapter is based on the channel-based exogenous coordination language Reo [Arb04] which allows a declarative definition of coordinators, called connectors, using either a graphical notation [Arb04] or a textual description language [DA18]. While the semantics of Reo has been initially defined in terms of timed data streams [Arb03; Arb04], several alternative characterizations have been presented in the literature [JA12]. *Constraint automata* (CA) [Bai+06] as well as their probabilistic counterparts, *probabilistic constraint automata* (PCA) and *simple probabilistic constraint automata* (SPCA) [Bai05],

can describe both the operational behavior of Reo connectors (including channels) and the interface behavior of the coordinated components. Like Reo, CA allow for a compositional construction of coordinators. Tool support for model checking of systems defined in terms of Reo and CA is provided by Vereofy [Bai+10].

The goal of this chapter is the transfer of the concepts introduced by the RBA approach to CA. This enables the compositional definition of role-playing coordinators in terms of CA and Reo. The analysis could then be carried out using Vereofy. However, since Vereofy does not support probabilistic model checking, we will pursue a different approach, as shown in Figure 7.1. The ReoCompiler [Reo] is a tool for generating code from a textual description of a Reo connector [DA18]. It has been extended with support for targeting the PRISM language with multi-actions which has already been presented in Section 6.2.2. The operational behavior of the components within the circuit can then be defined using either constraint automata or PRISM modules. This approach enables a quantitative analysis of role-based systems defined in terms of SPCA and Reo.

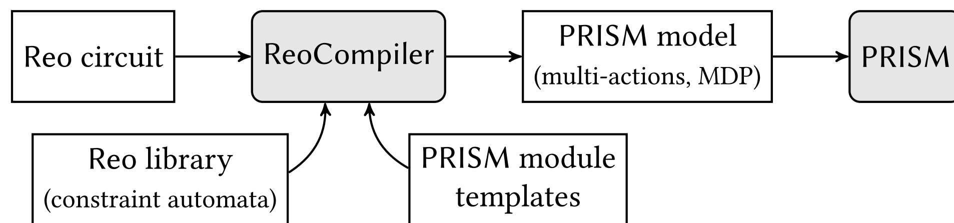


Figure 7.1: Using the extended ReoCompiler to generate PRISM models

Outline. This chapter is structured as follows. First, an overview of Reo is provided in Section 7.1, followed by the relevant definitions of CA and their composition. The embedding of role-related concepts into CA is discussed in Section 7.3. Finally, Section 7.4 presents notable details of the implementation, covering the extension of the ReoCompiler and further extensions of the PRISM language.

The content of this chapter is based on the publications [Chr+16a] and [Bai+18].

7.1 The exogenous coordination language Reo

This section provides a brief overview of Reo [Arb04] and its graphical notation. A Reo network, also called a *circuit*, consists of *components* and *connectors*. Connectors are built compositionally from simpler connectors where the simplest connectors are basic *channels*. The focus of Reo is on the connectors and the coordination of components. In the exogenous approach, components are not aware of each other or the environment they are used in. They communicate solely over one or more I/O ports (depicted as ∘) that allow them to send data or receive data over the connected channels.

Channels have two *ends*. Each end can be either a *source end* or a *sink end* which accept data into the channel or dispense data out of the channel, respectively. A channel is not necessarily one-directional and may have two source ends or two sink ends. Each channel has a type which specifies constraints for the data flow through the channel. Reo provides a set of standard channels:

Synchronous channel (○——▶○): The sync channel accepts data on its source end and atomically propagates it to its sink end. The channel only accepts data if the reader on its sink end is able to accept it, thus it synchronizes the reader and writer connected to its channel ends.

FIFO1 channel (○–□–▶○): This channel consumes a single data item at its source end and stores in a buffer. The data is later dispensed at the sink end once the connected reader is able to accept it.

Filter channel: A filter channel puts additional constraints on the incoming data, immediately disposing all data that does not pass the filter.

Synchronous drain (○▶——◀○): This channel has two source ends and consumes data items synchronously.

Asynchronous drain (○▶‖◀○): An asynchronous variant of the previous channel type which consumes data arriving at either end immediately. If two data items arrive at the same time, one of them is chosen nondeterministically.

Synchronous spout (○◀——▶○): The dual of the synchronous drain writes arbitrary data to its source ends, synchronously.

Asynchronous spout (○◀‖▶○): The asynchronous variant of the spout channel dispenses arbitrary data to one of the connected readers. If both readers are willing to accept data at the same time, one is chosen nondeterministically.

The set of basic channels can be extended with user-defined channels by defining components with two ports and the desired semantics.

Channels are composed by joining their channel ends in order to create more complex connectors. Joining channel ends results in a *node*. There are three different node types: A *source node* (Figure 7.2a) is formed by joining only sink ends and acts as a *replicator*. Any data that is written to a source node is synchronously written to all connected channel ends, but only if all channel ends accept simultaneously. In a *sink node* (Figure 7.2b), only sink ends are joined. This node type acts as a *merger* and only accepts data from exactly one of the coincident channels at a time. If multiple channels attempt to dispense data into the node at the same time, one is chosen nondeterministically. The combination of both source and sink nodes, the *mixed node* (Figure 7.2c), has a *merger-replicator* semantics. It nondeterministically accepts data from one sink end and synchronously dispenses it

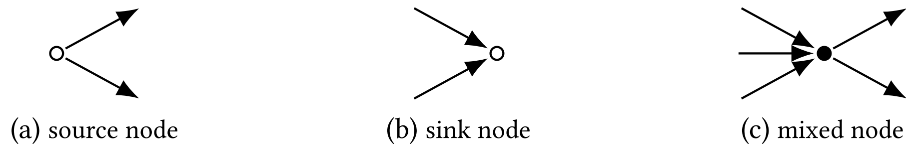


Figure 7.2: Reo node types

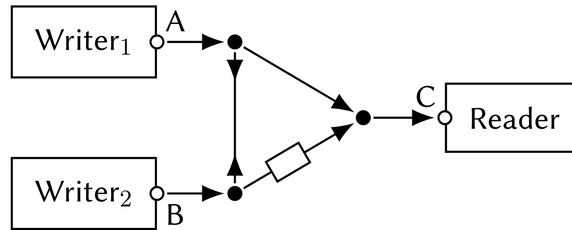


Figure 7.3: Two writers and a reader connected by an alternator connector

to all source ends. A variant of the standard mixed node is the router node (depicted as ⊗) which propagates data to exactly one nondeterministically chosen source end. The described behavior of Reo nodes allows for the construction of complex synchronization and coordination patterns by composing simple channels.

Example 7.1 (Alternator connector)**.** Figure 7.3 shows a connector that realizes an alternating data flow from two writers to a single reader. The synchronous drain synchronizes the connector's input ports A and B. While the data written to port A is immediately delivered to the output port C, the data from port B is stored in the FIFO1 channel. Note that the synchronous drain also prevents new data items from entering the connector as long as the FIFO1 is full. Once the data originating from port B is dispensed from the FIFO1 to port C, the process begins anew, resulting in the pattern ABABAB.... ■

CA [Bai+06] provide a compositional operational semantics for Reo connectors and can also be utilized to describe the I/O-behavior of components. Furthermore, the composition of CA matches the join operation of Reo for combining channel ends. An overview of CA and their composition is provided in the next section.

7.2 Constraint automata

CA [Bai+06] are a variant of transition systems where the transitions are labeled with data-dependent I/O-operations. They are suitable for describing the data flow in coordination models and thus provide an operational semantics for Reo connectors and components. The states of a CA stand for the possible configurations of a connector, e.g., the contents of FIFO1 channels, and the transitions represent the possible data flows at, e.g., I/O ports, nodes, and channel ends.

A CA is defined over a finite set $\mathcal{N}$ of names, standing for observable data-flow locations or ports. A transition in a CA is labeled with a subset of $\mathcal{N}$ and a data constraint. A data constraint is a symbolic representation of all possible assignments of data items to ports. Data constraints are formalized as propositional formulas over the atoms "$d_A = x$" with $A \in \mathcal{N}$ and $x \in Data$, where *Data* is a finite data domain. In the following, we assume that there is a global data domain *Data* that is shared by all ports. The set of data constraints is inductively defined by the following grammar, where $A \in \mathcal{N}$ and $x \in Data$.

$$g ::= true \mid d_A = x \mid g_1 \vee g_2 \mid \neg g$$

The set of all data constraints over data domain *Data* and names $N \subseteq \mathcal{N}$ is denoted as $DC(N, Data)$. We use the shorthand notation DC if the set of names and the data domain are clear from the context.

Definition 7.2 (Constraint automaton [Bai+06])**.** A *constraint automaton* is a tuple $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q^{init})$, where

- Q is a finite set of states,
- $\mathcal{N}$ is a finite set of names,
- $\longrightarrow \subseteq Q \times \mathcal{P}(\mathcal{N}) \times DC \times Q$ is the transition relation, and
- $Q^{init} \subseteq Q$ is the set of initial states.

A transition of a CA has the form (s, N, g, t). Intuitively, the automaton moves from state s to t, where I/O-operations are performed by the ports $A \in N$ which satisfy the data constraint g. Here, the set N is called the name-set and g the guard of the transition. We use the notation $s \xrightarrow{N,g} t$ for $(s, N, g, t) \in \longrightarrow$.

Example 7.3 (CA for a FIFO1 channel)**.** Figure 7.4 shows the CA for a FIFO1 channel with input port A and output port B. The data domain consists of the elements 0 and 1. The states of the CA represent the data item that is stored in the buffer and the initial state denotes an empty buffer. Consequently, the transitions are guarded by the data items that are written to or read from the buffer. ■

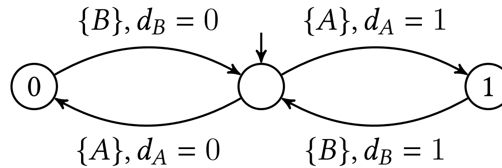


Figure 7.4: CA for a FIFO1 channel over data domain $Data = \{0, 1\}$ with input port A and output port B

The composition of CA formalizes both synchronous I/O-operations on shared ports and asynchronous I/O-operations. With that, the product operator on CA is adequate to describe the effects of joining channel ends in Reo networks.

Definition 7.4 (Constraint automata product [Bai+06]). The product of two CA $\mathcal{A}_i = (Q_i, \mathcal{N}_i, \longrightarrow_i, Q_i^{init})$ for $i \in \{1, 2\}$ is defined as

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 \;=\; \big(Q_1 \times Q_2,\, \mathcal{N}_1 \cup \mathcal{N}_2,\, \longrightarrow,\, Q_1^{init} \times Q_2^{init}\big)$$

where $\longrightarrow$ is the smallest transition relation fulfilling the rules shown in Figure 7.5.

$$(\mathrm{int}_1)\frac{s_1 \xrightarrow{N,g}_1 t_1 \qquad N \cap \mathcal{N}_2 = \varnothing}{\langle s_1, s_2 \rangle \xrightarrow{N,g} \langle t_1, s_2 \rangle} \qquad (\mathrm{int}_2)\frac{s_2 \xrightarrow{N,g}_2 t_2 \qquad N \cap \mathcal{N}_1 = \varnothing}{\langle s_1, s_2 \rangle \xrightarrow{N,g} \langle s_1, t_2 \rangle}$$

$$\mathrm{sync}\frac{s_1 \xrightarrow{N_1,g_1}_1 t_1 \qquad s_2 \xrightarrow{N_2,g_2}_2 t_2 \qquad N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle s_1, s_2 \rangle \xrightarrow{N_1 \cup N_2,\, g_1 \wedge g_2} \langle t_1, t_2 \rangle}$$

Figure 7.5: Rules for the product of CA

The CA product is both commutative and associative.

For capturing probabilistic channels and connectors, CA have been extended to SPCA and the more general PCA [Bai+06]. For SPCA, the transition relation allows for a probabilistic choice of the successor state, formally $\longrightarrow \subseteq Q \times \mathcal{P}(\mathcal{N}) \times DC \times Distr(Q)$. An example for a channel that can be modeled using an SPCA is a lossy FIFO1 channel where writing data into the buffer fails with a certain probability. On the other hand, a probabilistic lossy synchronous channel that drops data written to its source end with a certain probability cannot be adequately described using an SPCA since here the data flow itself is probabilistic. However, the more expressive PCA can capture such channels. Here, not only the successor state, but also the name set as well as the data constraint are chosen according to a probability distribution. Since SPCA are expressive enough for embedding RBA, we will not consider PCA in this chapter.

7.3 Embedding of role-based automata in constraint automata

The goal of this section is to transfer the concepts introduced by the RBA approach (Section 5.2) to CA and Reo. In particular, role-playing should be made explicit by providing appropriate annotations. Furthermore, the CA-based approach should allow a compositional construction of role-based systems. Thus, corresponding operators for parallel composition and role-binding must be provided. Most importantly, the approach should enable the coordination of role-playing using Reo connectors.

Analogous to the RBA approach, we will use CA as a uniform representation for naturals, roles, and compartments. A natural can be represented by a standard CA. For

capturing roles and role-specific behavior, the transitions of a role-CA must be annotated with its respective role name in order to make the role-playing explicit. The idea here is to treat some of the names in the CA as role names. Formally, we partition the set of names $\mathcal{N}$ into a set of role names R and other names $\mathcal{N}'$. Then, the name-sets of the CA transitions explicitly encode role-playing, i.e., if $r \in N$ for some name-set N and a role name $r \in R$, then the role r is played in this transition. Contrarily, if $r \notin N$, the role is explicitly not played. Like in the RBA formalism, compartments arise from a combination of multiple roles. With the basic building blocks in place, we can now turn to the role-specific composition of CA.

Remind that the parallel composition of RBA mainly corresponds to the standard parallel composition of MDPs, but additionally handles the role-playing annotations. Specifically, the role-playing annotations are combined upon synchronization, allowing that multiple roles are played in a single transition. Since role-playing is indicated by special role names in the name-sets of a CA, the same effect is achieved by using the standard product operator on CA. In the synchronization rule (see Figure 7.5), the union of the name-sets also combines the role-playing.

Next, we consider the second composition operator, role-binding. The role-playing annotations in an RBA serve a dual purpose. Not only do they indicate role-playing, they also denote that certain transitions are added to the player upon role-binding. Since the encoding of role-playing in CA described above only provides information about role-playing, we introduce additional labeling functions that indicate which transitions of the corresponding CA will be added to the player on role-binding. Formally, for each role r we define a function $\mathcal{O}_r\colon Q \times \mathcal{P}(\mathcal{N}) \times DC \times Q \to \mathcal{P}(R)$. If for some transition $s \xrightarrow{N,g} t$ with $r \in N$ we have $r \in \mathcal{O}(s, N, g, t)$, then the transition is added upon binding r. This corresponds to a "$+r$"-annotated transition in an RBA. The role-binding operator can now be adapted to CA.

Definition 7.5 (Role-binding for CA)**.** Let $\mathcal{A} = (Q_a, \mathcal{N}_a, \longrightarrow_a, Q_a^{init})$, $\mathcal{P} = (Q_p, \mathcal{N}_p, \longrightarrow_p, Q_p^{init})$ be CA and $\mathcal{O}_r$ be the labeling function for the role r. *Binding* an unbound role $r \in \mathcal{N}_a$ in $\mathcal{A}$ to a player $\mathcal{P}$ with $r \notin \mathcal{N}_p$ yields a CA

$$\mathcal{A} \bowtie_r^{\mathcal{O}_r} \mathcal{P} \;=\; \big(\, Q_a \times Q_p,\; \mathcal{N}_a \cup \mathcal{N}_p,\; \longrightarrow,\; Q_a^{init} \times Q_p^{init} \,\big)$$

where $\longrightarrow$ is the smallest transition relation fulfilling the rules shown in Figure 7.6.

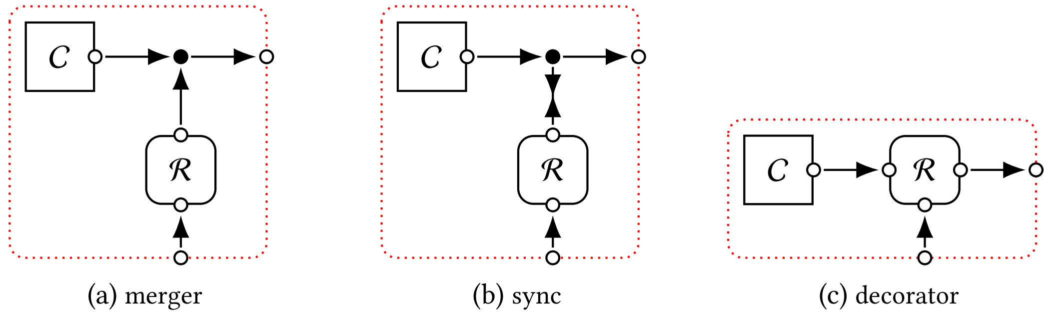
An alternative to the role-binding operator presented above is to utilize Reo for constructing a connector that binds a role component to a player component. The main advantage of this approach is that it enables using standard tools for modeling and analyzing Reo circuits also for role-based systems since no special role-binding operator is needed. As the full Reo framework can be employed to construct such a binding connector, the interactions and coordination between a player component and its role components can be arbitrarily complex.

$$(\mathrm{int}_a)\ \frac{s_a \xrightarrow{N,g}_a t_a \qquad N \cap \mathcal{N}_p = \varnothing}{\langle s_a, s_p \rangle \xrightarrow{N,g} \langle t_a, s_p \rangle} \qquad (\mathrm{int}_p)\ \frac{s_p \xrightarrow{N,g}_p t_p \qquad N \cap \mathcal{N}_a = \varnothing}{\langle s_a, s_p \rangle \xrightarrow{N,g} \langle s_a, t_p \rangle}$$

$$(\mathrm{sync})\ \frac{s_a \xrightarrow{N_a, g_a}_a t_a \qquad s_p \xrightarrow{N_p, g_p}_p t_p \qquad N_a \cap \mathcal{N}_p = N_p \cap \mathcal{N}_a \qquad r \notin O_r(s_a, N_a, g_a, t_a)}{\langle s_a, s_p \rangle \xrightarrow{N_a \cup N_p,\, g_a \wedge g_p} \langle t_a, t_p \rangle}$$

$$(\mathrm{add})\ \frac{s_a \xrightarrow{N,g}_a t_a \qquad N \cap \mathcal{N}_p \neq \varnothing \qquad r \in O_r(s_a, N_a, g_a, t_a)}{\langle s_a, s_p \rangle \xrightarrow{N,g} \langle t_a, s_p \rangle}$$

Figure 7.6: Rules for binding a role r within a CA $\mathcal{A}$ to a player $\mathcal{P}$



(a) merger (b) sync (c) decorator

Figure 7.7: Patterns for binding role components to player components. Roles components are depicted with rounded corners and the role-binding connector is surrounded by a dotted line. The bottom ports control role-playing.

Depending on how a role should adapt its player, different connectors may be appropriate. If a role only adds behavior, the connector shown in Figure 7.7a is sufficient. Here, the outputs of the player component C and the role component $\mathcal{R}$ are merged using a standard Reo node. Thus, $\mathcal{R}$ can neither block nor modify the output of C (assuming fairness of the merger). The case where the role acts solely as a filter and suppresses certain behavior is realized by the connector shown in Figure 7.7b. The synchronous drain channel forces the synchronization between C and $\mathcal{R}$ for every outgoing data item. Thus, $\mathcal{R}$ can inhibit certain outgoing messages by refusing to synchronize. Since the synchronous drain consumes the data sent by $\mathcal{R}$, the role cannot add new behavior. The connector shown in Figure 7.7c implements the most general case for role-binding. Here, the role component may suppress or modify any data sent by the player and can add new behavior as well. Clearly, this binding pattern subsumes both the patterns in Figures 7.7a and 7.7b. However, the fact that the role component in Figure 7.7a cannot modify or suppress the data sent by the player is apparent from the structure of the connector. This is not the case in Figure 7.7c where one would have to examine the operational behavior of

the role component to establish the same guarantee. Thus, by using behavior-restricting connectors for role-binding, certain guarantees can be established without having to take the components' implementations into account. The binding patterns shown in Figure 7.7 can be generalized easily to account for additional ports and roles.

The last missing piece of the CA-based framework for constructing role-based systems is the coordination of role-playing. Recall that a role r is actively played in a transition $s \xrightarrow{N,g} t$ if $r \in N$. This encoding implies that for each role there is an I/O-port which is active whenever the role is played. Thus, it is sufficient to provide a coordination component (also in the form of a CA) that allows, enforces, or blocks I/O-activity on these ports to coordinate role-playing. Naturally, this coordination component can be obtained by constructing a Reo connector. This approach enables a declarative and compositional description of role-playing coordinators. The role-oriented modeling framework presented in this section can be extended straightforwardly to SPCA which makes it as expressive as the RBA approach.

7.4 Implementation

This section presents a general approach for the exogenous coordination of stochastic components. The components are defined using PRISM's guarded command language and the coordination "glue code" is described using Reo. The implementation of this approach is twofold. First, PRISM has been extended with support for multi-actions and additional language constructs for component-based modeling. Second, the ReoCompiler [Reo] has been extended with support for PRISM as a new target language. This extension enables the automatic generation of a PRISM model from a textual description of a Reo circuit [DA18] that coordinates PRISM modules. Furthermore, support for the concept of reward monitors has been added to the ReoCompiler which allows attaching rewards and costs to port activity in the Reo network. This facilitates a quantitative analysis of Reo networks using PRISM's extensive analysis support. While the implementation presented in the following enables the analysis of arbitrary Reo circuits, it is in particular also applicable for the role-oriented modeling approach presented in the previous section.

7.4.1 Exogenous coordination of PRISM modules

The guarded command language of PRISM has been extended with language features that facilitate an exogenous coordination of modules. Most importantly, the PRISM language has been extended conservatively with support for multi-actions, i.e., a command can be labeled with a set of actions instead of only a single action. Multi-actions naturally correspond to the name-sets in CA and port names in Reo. The adapted parallel composition for modules with multi-actions has been derived from the data-abstract variant of the SPCA product. For details on the multi-action extension, see Section 6.2.2.

In addition to the multi-action extension, several language constructs that simplify exogenous and component-based modeling have been added. The standard PRISM language provides a mechanism for obtaining a copy of an already existing module by renaming all its variables and possibly renaming some of its actions. The renaming is necessary since all variable names are global in the PRISM language and thus must be unique. The main issue of this approach is that it requires detailed knowledge of the variable names within the module which complicates an automated creation of copies. For that reason, the language has been extended to allow a *rule-based renaming*. For instance, the following statement creates a copy of module `M1` named `M2` where the variable names in `M2` are obtained by adding the prefix `M2_` to all variable names of `M1`.

```
module M2 = M1 (varprefix=M2_) endmodule
```

Using this construct, the uniqueness of variable names can be guaranteed easily. Additionally, action names may be renamed using the `actionprefix` keyword. Alternatively, renaming may also append a suffix to variable names and action labels using the `varsuffix` and `actionsuffix` keywords, respectively. With this automatic renaming, the renaming statement can be seen as the *instantiation* of a module. However, in the standard PRISM language, every module appearing in the model file is instantiated automatically. In order to enable the definition of a library of modules that can be instantiated as needed, a module definition can be marked as a *template*. A module prepended with the `template` keyword is not instantiated automatically, but is available for instantiation via module renaming. The multi-action extension and the basic support for template instantiation make the PRISM language a viable target language for the ReoCompiler. This connection will be addressed in the following.

7.4.2 Reo for exogenous coordination within PRISM

For generating the coordination glue code for PRISM modules, we utilize the ReoCompiler tool developed at the Centrum Wiskunde & Informatica, Amsterdam [Reo]. Given a textual description of a Reo circuit [DA18], the ReoCompiler compiles the coordination glue code into a selected target language, such as Java. The generated code that implements the Reo connector can then be combined with the component implementations (e.g., provided as Java classes) which results in a complete implementation of the Reo circuit.

The ReoCompiler has been extended such that PRISM is one of the supported target languages. In particular, the extended compiler provides an automatic translation of the ReoCompiler's internal representation of connectors into the PRISM language with multi-actions. Together with the support for module instantiation from module templates, the generated model (given in the PRISM language) instantiates the PRISM modules referenced in the textual description of the Reo network and connects it with the generated coordination glue code. The connector and the PRISM modules interface via actions that have been exported as port names.

The semantics of the PRISM language with multi-actions corresponds to the data-abstract variant of SPCA. This is appropriate if the data domain of the Reo circuit is a singleton set. However, attaching data items to port activity is rather natural for some modeling tasks. In order to enable data-dependent actions in PRISM models, an additional tool has been developed which encodes data items into action names. For simplicity, the tool only supports data domains consisting of finite sets of integers. Consider the following example of a command where the action is treated as data:

```
[a] s = 0 & a = 1 -> (s' = 1);
```

Intuitively, this command states that the module can move from state `s = 0` to state `s = 1` if action `a` is executed and carries the data item 1. The tool translates the command by attaching the data item to the action name:

```
[a_1] s = 0 -> (s' = 1);
```

This tool enables the automatic translation of any data-aware Reo connector into the extended PRISM language.

Another extension of the ReoCompiler concerns cost and reward annotations on Reo circuits. This is useful, e.g., for specifying the energy consumption of certain components or for tracking the number of completed tasks. Rewards are captured by a special type of component called *reward monitor.* A reward monitor has one or more input ports that can be connected to the Reo network using channels as usual. The reward monitor definition then specifies the rewards that are assigned whenever certain input ports of the monitor are active. A reward monitor component is translated into a reward structure where rewards assigned to ports are straightforwardly transformed into transition rewards for the actions corresponding to the ports.

The extensions to the ReoCompiler and PRISM enable a quantitative analysis of systems comprising stochastic components that are coordinated by Reo connectors. This implementation is applicable for modeling and analyzing role-based systems with exogenous coordination, as will be demonstrated in a case study in Chapter 8.

8 Evaluation of the role-oriented approach

The evaluation of the role-oriented modeling approaches and the implementations is threefold. First, the practical applicability is demonstrated by means of three experimental studies. Second, the RBA approach is classified according to the characteristics of roles that have been identified by Steimann [Ste00] and Kühn et al. [Küh+14]. Third, the expressiveness of the approach is discussed by relating it to previous work for modeling adaptive systems.

8.1 Experimental studies

In this section, the applicability of the proposed modeling approaches is demonstrated by means of three experimental studies. The first two apply the RBA approach (see Chapter 5) and the role-oriented modeling language (see Chapter 6). The third experiment illustrates the potential of applying the exogenous coordination language Reo for defining the role-playing coordinator as proposed in Chapter 7. Furthermore, the scalability of the implementations and the translational approaches is evaluated.

The first experimental study, a peer-to-peer file transfer protocol, focuses on detecting unintended interactions originating from specific combinations of roles or the order of role-playing. The analysis considers both interactions within a single network and interactions between different networks. The second experimental study is an example from production automation, a self-adaptive robot production cell. Here, the quantitative impact of interactions is considered. Finally, the third experimental study again focuses on the peer-to-peer file transfer protocol, but this time utilizing a declarative definition of the coordinator as a Reo circuit.

The first two experimental studies are part of the evaluation in [Chr+20] and the third has been presented in [Bai+18].

Experiment setup. All experiments have been carried out on a system with two quad-core Intel Xeon L5630 CPUs (at 2.13 GHz) and 192 GB RAM running Debian 10.

8.1.1 Peer-to-peer file transfer

The peer-to-peer file transfer example has been adapted from [HK14]. The considered system comprises a number of computer systems called *stations* that constitute the naturals. Each station can store a number of files depending on its capacity. A station may play the roles *client*, *server*, and *relay* to request a file from any of its connected peers, to provide files to other stations, and to relay files and requests to neighboring stations, respectively. The model contains at least one *network* compartment that defines the topology of the system's network and specifies the file-transfer protocol. A station can be part of several networks. In that case, it has multiple distinct sets of server, client, and relay roles, one set of roles for each network. For instance, a station may concurrently play the client role in one network and the server role in another.

A file transfer is initiated by a station that assumes the client role and subsequently sends a request message. If another station that owns the requested file receives the request, it plays the server role. Otherwise, it plays the relay role and resends the request to its neighboring peers. A server fetches the file from its station and sends it to the client, possibly over one or more relays. Upon receiving the file, the client role stores the file on its corresponding station.

Modeling details

In the following, we will discuss how the role-oriented modeling approach and the modeling language facilitate modularization and a separation of concerns. For that, we examine notable details of the file-transfer model. The model is parametrized and can be instantiated for different network topologies, e.g., fully connected, star, and ring. The topology is specified in terms of unidirectional *links* between stations. The definition of a unidirectional ring topology with 3 stations is shown in Listing 8.1. Here, `links` is an array of links and a link consists of a source and a target station where each station is identified by its index. The definition states that station 0 can send a message to station 1 directly, but station 1 can only send a message to station 0 if the message is relayed by station 2. The extensive metaprogramming support of RML allows us to generically define the behavior of all components such that the model can be instantiated for any topology.

```
const links = [ [0, 1], [1, 2], [2, 0] ];
```

Listing 8.1: Definition of a unidirectional ring network topology

We now turn to the implementation of a station which is shown in Listing 8.2. This simplified version of a station can store a single file (line 2) which can be retrieved via the `load` action (line 5) and overwritten by the `store` action (line 6). Note that both actions

```
1  impl Station {
2    file : [0 .. NUM_FILES] init init_file[self.get_index()];
3
4    forall f : [1 .. NUM_FILES] {
5      [internal self.load[f]]  file = f -> true;
6      [internal self.store[f]] true -> (file' = f);
7
8      forall link : [0 .. LAST_LINK]. self.is_source_of(link) {
9        [req(f, link)]  false -> true; // send request
10       [data(f, link)] false -> true; // send data
11     }
12
13     forall link : [0 .. LAST_LINK]. self.is_target_of(link) {
14       [req(f, link)]  false -> true; // receive request
15       [data(f, link)] false -> true; // receive data
16     }
17   }
18 }
```

Listing 8.2: Module implementing a station

are marked as `internal` (see Section 6.1.2) since they are intended for the communication between a station and its roles only. Messages between stations over some `link` for a file `f` are modeled as actions `req(f, link)` and `data(f, link)` corresponding to request messages and file data messages, respectively. Here, `link` stands for an index to an element in the `links` array. The quantifier in line 8 ranges over all links that originate in the station, i.e., the links over which the station can send messages. Since a station which is not playing any role does not send messages by itself, the guards for both commands in lines 9 and 10 are `false`, thereby blocking these actions. Similarly, incoming messages are not accepted, thus the corresponding actions are blocked as well (lines 14 and 15). Modeling all messages as actions allows us to easily modify the behavior of a station by overriding specific actions.

The functionality of the client, server, and relay roles can be fully encapsulated within RML modules without having to modify the stations' modules in any way. As a representative example, we consider the implementation of the server role which is shown in Listing 8.3. A server has the three locations `S_IDLE`, `S_LOAD`, and `S_SEND` in which it waits for an incoming request, loads the file from its associated station, and sends the requested file over the network, respectively. An additional variable `buf` is used to temporarily store the file before it is sent. In order to react to incoming requests, the server role overrides the `req` actions (line 7). Analogously, the `data` actions for outbound messages are overridden to send the file to connected peers. The client and relay roles

```
1 impl Server {
2   loc : enum { S_IDLE, S_LOAD, S_SEND } init S_IDLE;
3   buf : [0 .. NUM_FILES] init 0;
4
5   forall f : [1 .. NUM_FILES] {
6     forall link : [0 .. LAST_LINK]. self.is_target_of(link) {
7       [override req(f, link)] loc = S_IDLE ->
8         (loc' = S_LOAD) & (buf' = f);
9     }
10
11    [core(self).load[f]] loc = S_LOAD & buf = f -> (loc' = S_SEND);
12
13    forall link : [0 .. LAST_LINK]. self.is_source_of(link) {
14      [override data(f, link)] loc = S_SEND & buf = f ->
15        (loc' = S_IDLE) & (buf' = 0);
16    }
17  }
18 }
```

Listing 8.3: Module implementing a server role

can be modeled similarly, again without modifying the module of the station or any of the other role modules. This clear separation of concerns provides extensive flexibility for instantiating various system variants. For instance, there may be stations that have no client role bound to them, i.e., they can only act as servers or relays. But more importantly, it allows us to bind a role of the same type multiple times such that a station may act as a server within different networks.

The model is translated into the standard PRISM language as detailed in Section 6.2. The MDP arising from the PRISM model has nondeterministic choices among all role-playings. Furthermore, the client issuing the next request is chosen nondeterministically as well. The MDP provides the basis for the following functional analysis.

Functional analysis

We first establish the functional correctness of systems consisting only of a single network. To be able to track progress within the system, a monitor component is added to the model. It contains Boolean variables g_f^s for all stations s and files f indicating whether station s has gotten file f at some point. These variables are initially *false* and set to *true* once station s stores file f. Further, we define the atomic propositions $send_c$ and $recv_c$ that hold in all states where the client role c has just sent a request and received the requested file, respectively. In addition to verifying that the model is free of deadlocks,

the following properties, given as CTL formulas, were checked. Here, S denotes the set of stations, C denotes the set of client roles, and F denotes the set of files.

1. Some station is able to get file A.
   $\forall\Box\exists\Diamond\big(\bigvee_{s\in S} g_A^s\big)$
2. All stations eventually received each file at least once.
   $\exists\Diamond\big(\bigwedge_{s\in S, f\in F} g_f^s\big)$
3. Each request will eventually be answered.
   $\bigwedge_{c\in C} \forall\Box\big(send_c \implies \forall\Diamond recv_c\big)$

Checking the above properties revealed several unintended role interactions that were subsequently ruled out by refining the model. In the following, we discuss two types of interactions in more detail.

```
1 coordinator {
2   forall f : [1 .. NUM_FILES] {
3     forall srv : Server {
4       forall link : [0 .. LAST_LINK] {
5         if srv.is_target_of(link) {
6           [req(f, link)] [ srv  ] station_of(srv).file = f  -> true;
7           [req(f, link)] [ !srv ] station_of(srv).file != f -> true;
8 } } } }
```

Listing 8.4: Coordinator specifying that the server role may only be played if its associated station possesses the file.

Interactions between a role and its player. The server role may be played for an incoming request even if its associated station does not possess the requested file (cf. line 7 of Listing 8.3). This eventually leads to a deadlock once the server tries to synchronize with its player over the `load` action (line 16). In order to prevent this unintended playing of the server role, the coordinator shown in Listing 8.4 is added to the model. The command in line 7 states that the server role must not be played in case its station does not have the file. Additionally, we must allow playing the server role in case the station actually has the file (line 6). Another interaction is caused by the client role. If a station has the last remaining copy of a file, its client role may overwrite it by requesting and receiving a different file. In this scenario a file is “lost”, and thus property (2) is violated. This fault can be prevented by extending the module of the network compartment such that requests can only be generated if the fulfillment of the request does not overwrite the last remaining copy of a file in the network.

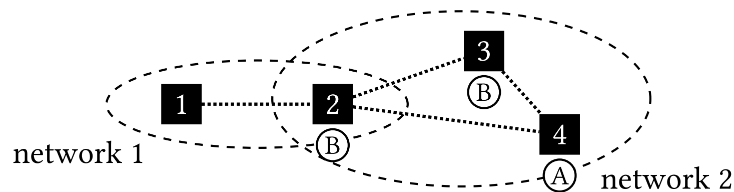


Figure 8.5: Overlapping networks with a shared station

Interactions between roles. Interactions between roles of the same player can occur if these roles override the same action(s). This is actually the case for the client and relay roles as they both override the `data` actions for incoming files. In case the client role has been requesting a file before, both the client and relay role can be played for the `data` action. However, playing the relay role in this scenario means that the file is sent along to another station instead of storing it. Again, this interaction can be prevented by extending the coordinator such that playing the client role is enforced for an incoming requested file. Unintended behavior may also emerge from interactions between roles of different players. The protocol only allows for a single file transfer at a time. Thus, if a client role issues a new request while another transfer is still in progress, the protocol is violated. This, in turn, may leave some components of the system in an intermediate state which ultimately leads to a deadlock. This issue is prevented by only allowing the generation of new requests if the network is currently idle.

With the refinements described previously all properties are satisfied, but only for systems that consist of a single isolated network. Due to complex interactions this is not the case for systems with multiple networks connected by shared stations. Consider the scenario shown in Figure 8.5 which comprises two network compartments, the first one containing stations 1 and 2, and the second one spanning over the stations 2, 3, and 4. Since station 2 is contained in both networks, it has two sets of server, client, and relay roles bound to it such that it can play these roles in either network. We assume that in this scenario stations 2 and 3 initially store file B and station 4 possesses file A. Checking the properties (1)–(3) on this model revealed the following unforeseen interactions.

While at most one transfer can happen within a single network at any given time, there may be concurrent transfers in different networks. This may lead to the following interaction. First, the client role of station 2 in network 2 requests and receives file A, but does not immediately store it on station 2. Likewise, the other client role of station 2 in network 1 requests and receives file B and also does not store it immediately. Then, the previously received file A is stored on station 2 by the client role in network 2. This now allows station 4 to receive file B and to overwrite its copy of file A. Finally, the client role of station 2 stores file B, thereby overwriting the last remaining copy of file A. From this point on property (1) can no longer be satisfied. Similarly, property (3) may be violated due to concurrent file transfers. Suppose station 1 decides to send a request for file B. Before actually sending the request, station 2 requests and receives file A in network

2, thereby overwriting file B on station 2. Then, station 1 actually sends its request to station 2. However, since station 2 no longer has file B, it will act as relay and send the request back to station 1. Because station 1 does not possess file B either, it will act in the relay role as well, and thus the process repeats indefinitely. This violates property (3) since the request of station 1 is never answered.

```
role-playing                                      action
client_0,                                         client_0_genreq_2
client_2,                                         client_2_genreq_1
client_0,                                         client_0_genreq_2
client_2,                                         client_2_genreq_1
client_2, not_relay_2, not_server_3, relay_3,     r_2_3_1
not_client_3, not_relay_4, relay_3, server_4,     r_3_4_1
server_4,                                         station_3_load_1
not_client_3, not_relay_4, relay_3, server_4,     d_4_3_1
client_2, not_relay_2, not_server_3, relay_3,     d_3_2_1
client_1, client_2,                               station_1_store_1
client_0, not_relay_0, not_server_1, relay_1,     r_0_1_2
not_client_1, not_server_0, relay_0, relay_1,     r_1_0_2
not_client_0, not_server_1, relay_0, relay_1,     r_0_1_2
not_client_1, not_server_0, relay_0, relay_1,     r_1_0_2
```

Listing 8.6: Action trace for the violation of property (3)

Violations of the properties caused by unintended interactions are reported by PRISM in the form of a counterexample. Listing 8.6 presents the action trace for the violation of property (3) in the two-network scenario (Figure 8.5). Each line corresponds to a single system state that has been reached by executing the action in the second column while the roles in the first column were played or not played. For better readability the system states have been omitted and the trace has been reformatted, but otherwise appears verbatim. The last two lines in the trace form a loop corresponding to the repeated relaying of the request message. Since all role-playing is encoded into the actions of the MDP, tracing errors back to the original role-based model is straightforward. Furthermore, the explicit representation of role-playing in the counterexample trace helps to identify the relevant roles in the interaction.

In summary, the functional analysis revealed that network-local coordination alone is not sufficient in case of interacting networks with shared stations. In order to prevent the described interactions the coordinator could be extended by additional global constraints over all networks. For instance, concurrent file-transfers involving shared stations could be disallowed.

Scalability

The file transfer model can be instantiated for any number of files and stations which allows us to investigate the scalability of the analysis approach. More precisely, the model has been instantiated for star topologies where each station has a bidirectional connection to the center station. To determine the impact of the role-oriented modeling as well as the translational analysis approach, a second file-transfer model using only standard PRISM language constructs has been created. This standard PRISM model contains neither any role-binding nor a coordinator[1]. Note that the standard model is tailored to the specific benchmark scenario and is less flexible and less modular than the role-based model. Adding a second network to the system, for instance, would require extensive rewriting of the model.

The model sizes for various numbers of stations and files is shown in Table 8.1. The size is listed both in terms of the number of reachable states and the number of nodes in the MTBDD that PRISM uses to represent the MDP symbolically. The number of MTBDD nodes has been minimized via sifting, a heuristic reordering of the MTBDD variables [Kle+18], for each model instance separately. The table also lists the time required for checking the properties described previously. The analysis time has been averaged over three runs with an additional warm-up run beforehand. The time for translating the RML model into a PRISM model is not included in the build time, as it accounts for less than 1% of the overall analysis time. Even for the largest instance the translation took only 1.2 s on average.

For visualizing the overhead introduced by the role-based approach, Figure 8.7 shows the analysis time of the role-based models relative to the analysis time of the standard model. If a point is above the 1-line, the role-based analysis is slower than the analysis of the standard model. For instance, analysis of the instance with 4 stations and 3 files takes roughly thrice as long. The overhead of the role-oriented approach is caused by the additional role-playing actions present in the MDP that also have to be encoded in the MTBDD. Furthermore, the multi-action extension of PRISM [Bai+18] uses a different encoding of actions than standard PRISM which causes further overhead in the form of additional MTBDD variables. Nevertheless, for larger instances the impact of this overhead diminishes. It turns out, for the larger instances with 2 or 3 files, the structure of the role-based model is favorable for the conducted analysis and leads to an overall lower analysis time compared to the standard model. In summary, the experiments showed that the role-oriented analysis approach is feasible in practice, even for larger models.

[1] To be able to freely scale the standard PRISM model, it has been defined in RML as well. However, only the metaprogramming constructs of RML have been utilized in order to generically define the model. In principle, any other template language could have been used as well.

Table 8.1: Model sizes, build times, and analysis times for increasing numbers of stations and files in the file-transfer model

			Role-oriented model			Standard PRISM model		
St.	Files	States	Nodes	Build (s)	Analysis (s)	Nodes	Build (s)	Analysis (s)
2	1	20	2 401	0.114	0.016	384	0.047	0.017
3	1	83	10 810	0.352	0.030	1 540	0.076	0.032
4	1	328	24 873	0.760	0.063	3 773	0.124	0.073
3	2	981	45 127	1.368	0.129	5 533	0.221	0.150
5	1	1 063	44 900	1.552	0.143	7 229	0.225	0.166
6	1	3 126	69 837	2.874	0.261	11 433	0.362	0.299
7	1	8 625	101 313	4.847	0.415	17 506	0.624	0.383
4	2	13 937	98 575	3.702	0.483	13 646	0.745	0.608
8	1	22 748	138 624	6.997	0.654	24 054	0.923	0.807
9	1	58 007	185 849	10.658	1.172	34 362	1.581	1.628
5	2	130 779	177 263	9.245	1.596	29 539	2.598	1.815
4	3	235 843	208 195	9.494	1.525	23 171	2.230	1.612
6	2	1 035 819	281 856	17.535	5.299	64 419	7.862	5.513
7	2	7 450 395	419 064	35.844	13.237	134 222	21.522	17.964
5	3	8 770 909	381 840	27.208	13.111	69 169	13.079	13.771
8	2	50 362 803	593 018	65.928	31.791	200 442	51.566	41.718
5	4	193 690 961	683 121	93.195	86.554	123 723	66.381	97.930
6	3	222 866 842	668 929	126.077	119.330	226 758	164.400	163.973
9	2	326 024 459	884 738	133.810	83.126	411 685	147.653	115.856
7	3	4 722 943 030	1 114 291	544.446	852.297	793 286	1527.186	1911.909

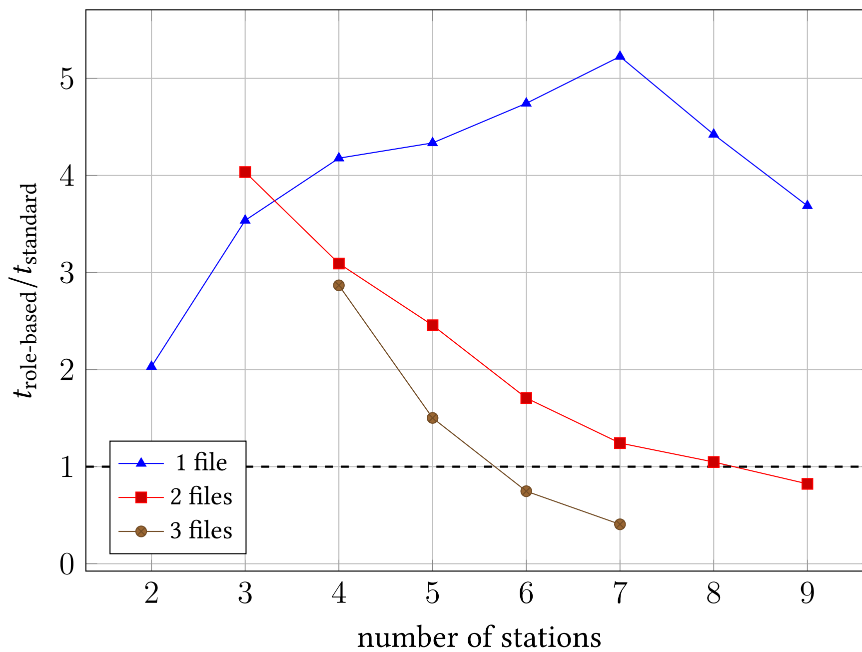


Figure 8.7: Analysis time of the role-based models relative to the standard models for fixed number of files

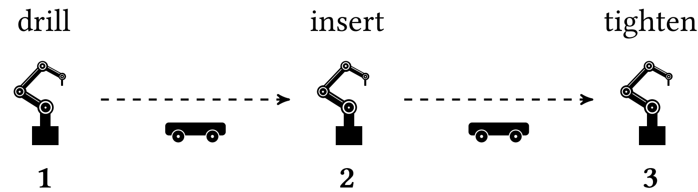


Figure 8.8: An automated production cell consisting of three robots connected by two autonomous carts

8.1.2 Self-adaptive production cell

The second experimental study is an example from production automation, an autonomous production cell that is able to adapt itself in case of failures (adapted from [GOR06]). A production cell consists of robots and a number of autonomous carts that transport workpieces between the robots. Each robot is equipped with a tool to fulfill a certain task. An example of a cell with three robots is shown in Figure 8.8 where the first one drills holes, the second one inserts screws, and the third one tightens them. Robots can switch their equipped tool, thus each robot is basically able to perform each task. However, it is assumed that switching tools takes a considerable amount of time. Therefore, a sensible configuration of the production cell assigns exactly one of the tasks to each robot. Each time a tool is used it may break with a given fixed probability. If a workpiece cannot be processed further because of a broken tool, the cell is reconfigured to restore its ability to process further workpieces.

Modeling details

Similar to the work by Güdemann et al. [GOR06], the cell has been modeled using a role-oriented approach. The role of a robot determines its assigned tool, e.g., the robot playing the *driller* role is responsible for drilling holes. This approach enables a separation of concerns in two ways. First, the tools' functionality can be encapsulated and modularized, and second, the reconfiguration of the production cell can be accomplished by changing the role assignment. Both are reflected and implemented in the RML model, i.e., the reconfiguration logic is fully contained in the coordinator. The functional correctness of the model has been established, i.e., it has no deadlock states and workpieces are processed fully and in-order. This provides the basis for the following quantitative analysis.

Quantitative analysis

The first analysis goal is to quantify the benefit of employing a self-adaptation mechanism for the resiliency of the overall system. For this, we consider three variants of the production cell with three robots. The *fixed* variant possesses no self-adaptivity and provides a baseline to compare the adaptive variants to. The *adaptive-chain* and *adaptive-*

ring variants are able to self-adapt with varying degrees of flexibility. In the *adaptive-chain* variant, only the direction in which the workpieces travel through the production cell can be changed, resulting in two possible configurations. The *adaptive-ring* variant allows that carts move directly between robots 1 and 3, thus each robot can potentially assume any role. In the following experiments, we assume that the probability that a tool breaks upon usage is 0.01. We consider the maximal probability that n workpieces are processed fully. A higher probability corresponds to a more resilient system. The results are presented in Figure 8.9. The plot clearly shows that introducing self-adaptivity significantly increases the resiliency of the system proportional to the flexibility provided to the coordinator.

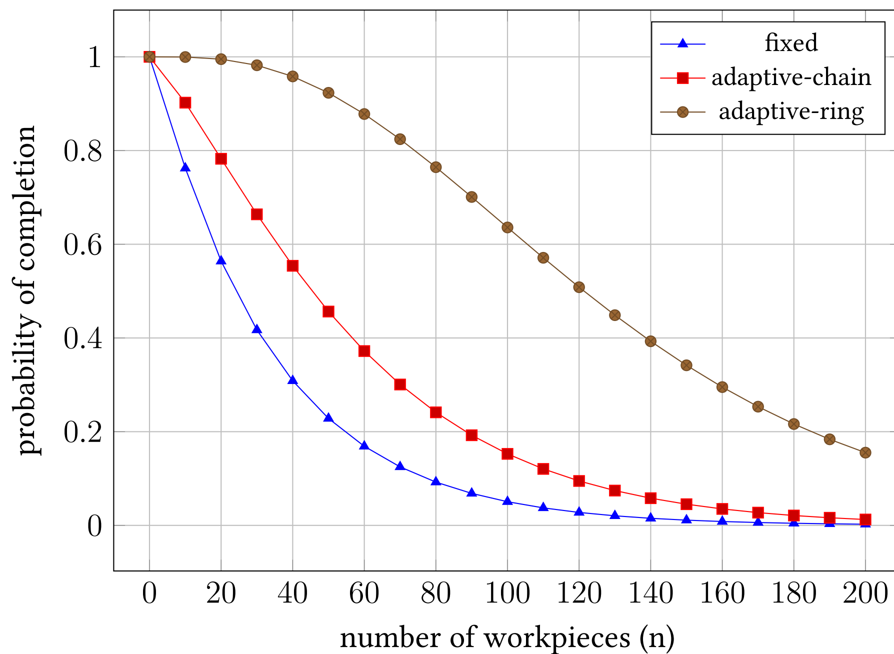


Figure 8.9: Maximal probability of fully processing n workpieces

Next, we consider a scenario where one of the robots can process workpieces twice as fast as the others. To fully utilize its additional productivity, the robot may be shared between two production cells. This allows us to increase the throughput of the production without adding a complete second production cell. A possible setup is shown in Figure 8.10. The throughput is defined as the number of finished workpieces per time unit where each processing step and each reconfiguration of a production cell takes one time unit. Note that robots can work in parallel. For instance, robots 1 and 3 may be drilling into two different workpieces within the same time unit. In order to reason about the throughput in the model, a reward of 1 is assigned to each transition that completes a workpiece. An additional monitor module keeps track of the elapsed time by incrementing a counter in each processing and reconfiguration phase. In case of multiple production cells within a single system we consider two possible adaptation schemes. A *localized* adaptation

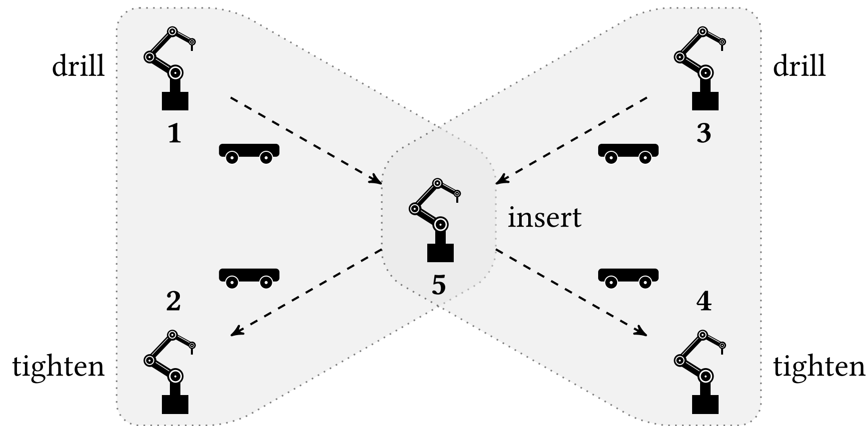


Figure 8.10: Two automated production cells sharing a robot

Table 8.2: Expected throughput of production cells

Production cells	Adaptation	Expected throughput min	max
1	localized	0.2531	0.2827
2 (shared robot)	localized	0.3493	0.5096
2 (shared robot)	global	0.4035	0.5474
2	localized	0.6870	0.7964

is limited to the robots of a single production cell while a *global* adaptation potentially reconfigures all robots in the system. From now on, we assume that the probability for a tool breaking is 0.1. To quantify the increase in throughput, we compare a single production cell with a system consisting of two overlapping production cells as shown in Figure 8.10. The results are shown in Table 8.2. For reference, the throughput of two individual production cells is shown in the last line. Both in the best and the worst case, the shared-robot variant has a significantly higher throughput than a single cell while only requiring two additional robots. The throughput of two cells is more than double than that of a single cell which is caused by the increased redundancy of the overall system. That is, even if one of the two cells fails completely, the other one might still be able to process workpieces.

Adding a second overlapping production cell increases the overall throughput. However, the shared robot may cause unintended interactions between the production cells. Since it can only be equipped with one tool at a time, the tool assignment for the shared robot has an influence on both production cells. Suppose that in the scenario shown in Figure 8.10 the *drill* tool of robot 1 breaks. To further process workpieces the left cell adapts itself by assigning the *drill* tool to robot 5 and the *tighten* tool to robot 1. The routes of the automated carts are adapted accordingly. But then, there is no longer any robot

responsible for *inserting* in the right cell. Thus, the next time a screw needs to be inserted the right cell might reassign the *insert* tool to the shared robot. However, then the left cell is required to adapt again once the next hole needs to be drilled which is causing an adaptation in the right cell once more, and so on. This interaction of frequent conflicting reconfigurations does not influence the functional correctness of the system. Workpieces are still fully processed by both production cells. However, as the reconfiguration of robots takes a certain amount of time, the throughput of the overall system is reduced in this case. The conflicting adaptations can be avoided by adopting a global adaptation scheme that takes all robots into account and keeps all cells operational if possible. For instance, in the scenario where the *drill* tool of robot 1 breaks, a global adaptation may assign the *drill* tool to robot 5 and the *insert* tool to *both* robot 1 and 3. This way, the processing capability of the left cell is restored and the subsequent conflicting reconfiguration of the right cell is avoided. The impact of the described interaction can be quantified by comparing systems with localized and global adaptation mechanisms. The results are presented in the second and third line of Table 8.2. Even though the global adaptation scheme requires a more sophisticated reconfiguration mechanism and coordination between different production cells, it significantly increases the throughput of the overall system.

8.1.3 File transfer with exogenous coordination

In this section, we revisit the peer-to-peer file transfer system from Section 8.1.1 to evaluate the approach presented in Chapter 7. In particular, we utilize the exogenous coordination language Reo to define the role-playing coordinator and leverage the presented PRISM extensions and Reo tool support for a quantitative analysis. In order to simplify the model and the analysis, only a single isolated network with either a ring topology or a chain topology is considered.

Modeling details

The behavior of the server, relay, and client roles is encapsulated within individual role components that are bound to the corresponding station components. Role-binding between a station component and its role components is achieved by constructing a binding connector as proposed in Section 7.3. Figure 8.11 depicts the binding connector in detail. The station component as well as the role components are modeled as PRISM modules. Each of the role components has one port that allows enabling or disabling the role by providing or not providing a data item to this port, respectively. The channels between the station component and the role components enable each role to retrieve the file stored on the station and to replace it with another one. The right part of the connector attaches the roles to the network (via the `in` and `out` ports), allowing each of them to send or receive messages, i.e., request messages or file data messages. The binding connector is the same for each station and is instantiated as needed. The network itself is

also realized as a Reo connector which connects the in and out ports of the stations and their respective roles according to the network topology.

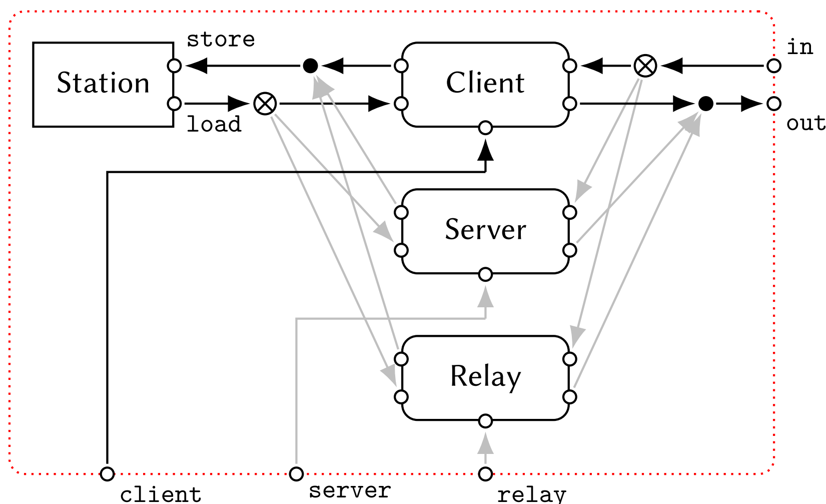


Figure 8.11: A station component and its bound role components

The client, server, and relay ports of a station's roles allow the role-playing coordinator to enable, enforce, or prohibit role-playing. In contrast to the monolithic coordinator of the model presented in Section 8.1.1, here the coordinator is split into multiple local coordinators and a single global coordinator. There is a local coordinator for each station and its bound roles. This local coordinator enforces that the Server role is only played if the station has the requested file. It also gives priority to the Client role over the Relay role in case the requested file arrives. These simple rules are sufficient to guarantee a correct execution of the file-transfer protocol. The sole responsibility of the global coordinator is to ensure that only a single transfer happens at any given time. The choice of the station which is allowed to send the next request is either random (using a uniform distribution over all stations) or nondeterministic. Overall, in this model the exogenous coordination approach enables a compositional definition of the role-playing coordinator using simple connectors with a localized responsibility. This enables a separation of concerns also for the coordinator and improves comprehensibility of the coordination compared to the monolithic approach.

Quantitative analysis

We consider the file-transfer model with three stations in four variants: using a chain topology or a ring topology, and using nondeterministic choice or probabilistic choice of the station sending the next request. For the quantitative analysis, reward monitors and state rewards have been added to the model. Network activity, i.e., the activity of at least one in or out port, consumes *energy*. Furthermore, a *penalty* is associated with

Table 8.3: Analysis results for the file-transfer model with 3 stations

		Query				
Topology	Scheduling	c	d	e	f	g
chain	nondet.	4.00	16.15	0.95	0.98	10.0
ring	nondet.	2.00	15.94	1.00	0.96	12.0
chain	random	5.87	18.73	0.64	0.65	15.0
ring	random	4.00	18.34	0.78	0.72	12.0

pending requests that have not yet been processed. A request is pending if a station wants to initiate a file transfer but has not been chosen by the global coordinator yet. After generating the model variants using the ReoCompiler, they have been analyzed using PRISM, asking for

(a) the minimal/maximal probability that eventually station 1 receives its requested file,
(b) the minimal/maximal probability that eventually all stations have received a file,
(c) the minimal expected time until the file requested by station 1 is delivered,
(d) the minimal expected time until all stations have received a file,
(e) the maximal probability to deliver a file to station 1 with less than x penalty,
(f) the maximal probability for delivering a file using a given energy budget without overstepping the penalty threshold, and
(g) the minimal energy required such that a file is delivered to station 1 with a probability greater than 0.9 without a penalty violation.

The analysis results for the queries (c) to (g) are shown in Table 8.3. As expected, the results for (c) show that in the ring topology the requested file is delivered faster since a direct transfer between stations is always possible. The difference between the optimal scheduling of requests and a random scheduling is shown by the results for (e). Generally, the random scheduling incurs a higher penalty as requests are kept waiting longer. The reward-bounded reachability probability (f) and the quantile [Bai+14] query (g) illustrate the trade-off between early processing of a request, or waiting for another request which causes a penalty for pending requests.

Scalability

Table 8.4 shows the model sizes and analysis times for the queries (f) and (g). The model sizes are given in terms of reachable states, the number of MTBDD nodes for representing the model symbolically, and the number of unique action names in the generated PRISM models. Compared to the file-transfer model presented in Section 8.1.1 (Table 8.1), the model with exogenous coordination of role-playing is considerably larger. Within the exogenous approach, the coordinator is not able to access the local state of the other components directly. Thus, the coordinator must obtain the necessary information by

Table 8.4: Model sizes and analysis times for the file-transfer model with 3 stations

					Time (s)		
Topo.	Sched.	States	Nodes	Actions	Build	Analysis f	Analysis g
chain	nondet.	4 204	38 870	150	10.7	81.1	58.4
ring	nondet.	34 164	97 471	150	90.7	197.8	91.3
chain	random	12 612	40 735	154	10.6	92.0	24.4
ring	random	102 492	102 326	154	62.6	224.5	101.0

reading data from the corresponding ports and then store it in its own local state. The additional message passing and the redundant storage of state information causes a much larger state space. Another issue is the large number of unique action names in the generated models. Every data-flow location that serves as an interface between a Reo connector and a PRISM module must have its own unique name. Since all complex components are modeled as PRISM modules, the number of these interface ports is quite large even for small instances of the model. The presented tooling currently does not apply any optimizations to reduce the number of action labels and the multi-action version of PRISM is not optimized for handling models with many action labels.

In conclusion, the exogenous modeling approach enables a compositional and declarative definition of role-playing coordinators. However, the current implementation of the approach is considerably less scalable than the RBA approach. The experiments in this section have shown that the exogenous modeling approach is viable, but could potentially benefit from further optimizations.

8.2 Classification

In order to evaluate the adequacy and expressiveness of the role-oriented modeling approach presented in this thesis, we check it against the properties of roles identified by Steimann in his survey of role-based approaches [Ste00]. Since modeling of systems is done in RML for the most part instead of using RBA directly, we will discuss the formalism and the modeling language in conjunction. Note that the notion of an *object* will be used synonymously with *natural* (instance) as the presented approach is not based on object-oriented concepts.

1. *A role comes with its own properties and behavior.*
   As roles are represented by an automata-based formalism, RBA, they can have their own state and behavior. Similarly, within RML a role is implemented using role modules whose state space is given by a set of variables and whose behavior is defined in terms of guarded commands.

2. *Roles depend on relationships.*
   This property is not enforced within the RBA approach as roles may exist without a counter-role, and thus can also represent different states or phases of their respective player.

3. *An object may play different roles simultaneously.*
   Multiple roles can be bound to the same player via nested role-binding (cf. Section 5.2.1). By means of synchronization over shared actions, roles of the same player may even be played simultaneously within the same transition.

4. *An object may play the same role several times, simultaneously.*
   Playing the same role type several times amounts to creating multiple copies of the role's RBA and binding them to the same player, again, via nested role-binding.

5. *An object may acquire and abandon roles dynamically.*
   The intended meaning of this property refers to the capability to dynamically switch between exposing role-specific behavior and not exposing this behavior. In the RBA formalism, therefore, acquiring and abandoning roles correspond to playing and not playing the role, respectively. Since playing a role is a property of a transition, switching between playing and not playing a role is achieved by choosing the respective transition with the desired role-playing annotation.

6. *The sequence in which roles may be acquired and relinquished can be subject to restrictions.*
   As described in the previous point, acquisition and removal of a role correspond to playing and not playing it, respectively. Role-playing may be restricted using the coordinator (see Section 5.2.3). Since the coordinator can have its own state, restricting the sequence of role-playing is easily accomplished.

7. *Objects of unrelated types can play the same role.*
   As long as the role and player interfaces are compatible, the role-binding operator poses no further requirements to the player. Thus, a role may be bound to players of different types.

8. *Roles can play roles.*
   A role can be bound to any RBA. Therefore, the player may also be another role which allows that roles can play roles.

9. *A role can be transferred from one object to another.*
   This is not directly supported, neither by the formalism nor by the role-oriented modeling language. However, a role transfer can be emulated by binding copies of the role to both the source and the target of the transfer. Then, the role of the source is played before the transfer happens and the role of the target is played after the

transfer. The state transfer between the copies can be modeled straightforwardly in RML by copying the local variables' values from the source role to the target role.

10. *The state of an object can be role-specific.*
If the RBA resulting from binding a role to a player is regarded as a compound object, then this object's state can indeed be role-specific.

11. *Features of an object can be role-specific.*
A role may override the behavior of its player or add new behavior. Therefore, the features of an object may be provided or modified by a played role.

12. *Roles restrict access.*
The concepts of visibility and accessibility of actions, state, and behavior do not exist within the RBA approach. Therefore, binding a role cannot restrict access. However, blocking actions of the player may be considered to be an access restriction.

13. *Different roles may share structure and behavior.*
This property refers to the concept of inheritance present in many object-oriented languages. Neither RBA nor RML support the inheritance of behavior.

14. *An object and its roles share identity.*
This property is not directly applicable to RBA because object identities are not a first-class concept within the formalism. However, since binding a role to a player yields a single RBA that incorporates both the role's RBA and the player's RBA, it can be argued that they have a shared identity.

15. *An object and its roles have different identities.*
It follows directly from the previous point that this property does not hold for the RBA formalism. However, in RML instances of roles and players can be distinctly addressed by their name to refer to their local variables and actions. In that sense they do not have a shared identity.

The previously listed properties of roles mainly concern the behavioral aspects of roles and the notion of relationships between roles. In a more recent survey by Kühn et al. [Küh+14], a shift of contemporary approaches to context-dependent roles has been identified and the list of role properties has been extended to account for the additional characteristics.

16. *Relationships between roles can be constrained.*
Neither the RBA formalism nor RML provide first-class relationships, therefore, no constraints on relationships can be defined.

17. *There may be constraints between relationships.*
For the same reason as in the previous point, defining constraints between relationships is not possible.

18. *Roles can be grouped and constrained together.*
    There is no explicit support for role groups. However, RML provides a `count` function to count the number of role instances of a certain type within a compartment. When used within the `system` block, cardinality constraints for a set of roles can be defined.

19. *Roles depend on compartments.*
    Compartments are a first-class concept in both the RBA formalism and in RML. While roles can be part of a compartment, i.e., depend on it, they may also exist without one. In RML the property that each role is contained within a compartment can be enforced by adding the following constraint to a `system` block.

    ```
    forall r : role. exists c : compartment. r in c;
    ```

20. *Compartments have properties and behavior.*
    Like roles, compartments are represented by RBA and, therefore, this property holds for them as well.

21. *A role (type) can be part of several compartments.*
    In RML it is possible to refer to the same role type in different compartment-type definitions and even put different constraints on them. The property is not applicable to the pure RBA formalism since it does not provide a type level.

22. *Compartments may play roles like objects.*
    This is possible since compartments are also represented by RBA which allows us to bind roles to them.

23. *Compartments may play roles which are part of themselves.*
    The RBA of a compartment arises from the parallel composition of the RBA for the roles contained in the compartment. Binding a role within the compartment to the compartment itself means that the compartment's RBA would need to be present in both positions of the binding operator which is not possible without creating a copy of the compartment.

24. *Compartments can contain other compartments.*
    The nesting of compartments, e.g., a *university* compartment containing a *faculty* compartment is not possible in the presented approach.

25. *Different compartments may share structure and behavior.*
    As is the case for roles, the inheritance of compartment behavior is not supported.

26. *Compartments have their own identity.*
    This property is not applicable for the RBA formalism as it does not include first-class identities. However, in RML, compartments are a first-class concept and have their own identity.

27. *The number of roles occurring in a compartment can be constrained.*
    RML directly supports the specification of occurrence constraints as part of the compartment-type definition.

The role-oriented modeling approach presented in this thesis covers the key characteristics of roles and includes compartments as an integral part of the formalism. With that even more advanced properties of roles, e.g., that compartments themselves can play roles, are fully supported. Neither the RBA formalism nor the modeling language provide an explicit notion of relationships. Rather, they are implicitly defined by the shared actions between roles. Nevertheless, the FOL-based constraint language of RML is expressive enough to mimic the effect of relationship constraints even without an explicit notion of relationships. Another aspect that is not covered by the approach is the inheritance of role behavior and compartment behavior. However, this is not surprising given the level of abstraction in RBA. Furthermore, formal models that lend themselves to an analysis using (probabilistic) model checking usually do not reach a scale where inheritance would bring a significant benefit to modeling. To summarize, the above classification as well as the case studies show that the approach is adequate for modeling role-based systems.

8.3 Related work

As the related work on role-based systems that inspired the approach in this thesis has already been discussed in Section 5.1, we focus here on related work regarding the formal modeling as well as the formal analysis of role-based systems. In particular, we discuss approaches that allow for a compositional modeling of adaptive systems and a modularization of adaptations. The body of work tailored specifically to role-based systems is rather limited. Therefore, we also consider approaches for aspect-oriented systems and feature-oriented systems, since the adaptation techniques used there are conceptually similar to those applied for role-based systems. In order to demonstrate the expressiveness of the RBA-approach presented in this thesis, we also discuss possible embeddings of related approaches.

8.3.1 Role-based approaches

The *Objects with Roles* [Per90] approach employs the role concept to capture the evolving behavior of objects. Within this approach, objects may play one or more roles simultaneously, even multiple roles of the same type. Only the behavioral aspect of roles is addressed, i.e., neither role relationships nor context-dependent roles are considered. The behavior of objects and roles is specified uniformly using *state transition rules* over states, roles, and messages. Similar to guarded commands, a rule may specify a guard in the form of a required source state and optionally an incoming message. If the conditions of a rule are satisfied, it may update the local state, instantiate or destroy roles, and send

a message. Given an initial state, repeated application of the rules induces a transition system with message-labeled edges, called *role state diagrams*. The concept of a *complex abstract state* of an object is introduced which contains the object's own local state as well as the state of each role the object plays. If a role is added to an object, the state transition rules of the object and the role are joined. In any given complex abstract state there is a nondeterministic choice among the applicable rules. Therefore, object and role behaviors may run concurrently (via interleaving). A role may adapt the behavior of an object using special rules that inhibit certain messages or even remove whole rules from the object's set of rules. By removing rules and adding new ones in their place, a modification of object behavior is possible. This is similar to the approach within RBA, where certain transitions of the player can be blocked and new transitions are added instead. In addition to the state transition rules, a model may also include *integrity* rules to define invariants. However, the approach makes no assumptions on how the violation of the integrity rules is handled. In particular, the paper focuses purely on the modeling aspect and no verification approach is presented.

The *superimposition* control structure introduced by Katz [Kat93] facilitates the modularization of distributed (reactive) systems, but is general enough to be applicable to other kinds of systems as well. The construct specifically targets the modularization of cross-cutting concerns such as the fulfillment of a common (temporary) subtask that requires multiple processes to collaborate. A superimposition declaration comprises the basic processes and several *role types* that capture the collaboration-specific behavior of a process. Using a *combination* operator, a role type of a superimposition is composed with a basic process. The role and the basic process operate on the same local state. A role type may define additional behavior, but is also able to adapt the existing behavior of the basic process. For that, it may declare *rewrite rules* that operate directly on the *abstract syntax tree* (AST) defining the process behavior which enables arbitrary modifications. This makes the superimposition construct quite general and allows it to be integrated into various modeling and programming languages. A notable instance is its use within feature-oriented systems as in the approach of Plath and Ryan [PR01]. Here, the superimposition construct is applied to integrate feature behavior into the base system. The construct is tailored to the input language of the model checker NuSMV and allows for two kinds of rewrite rules. Using the `treat` keyword, each read access to a variable can be replaced by another expression (which may involve the variable itself). The `impose` keyword, on the other hand, allows the modification of every write access, i.e., update, of a specific variable. Both kinds of rewrites can also be applied conditionally depending on the system's state. In contrast to the binding operator on RBA which operates directly on the transitions of the RBA, the combination operator of the superimposition construct modifies the behavior on the level of rules or guarded commands (which is also the case for the previously mentioned Objects with Roles approach). The consequence is that within the RBA approach only the observable behavior of the player, i.e., its actions, can be

modified, while superimposition also allows for arbitrary changes of the internal behavior. Note that this requires knowledge about the implementation of the adapted process which may lead to a tighter coupling between the role type and the adapted process. Another prominent application of superimposition is delta-oriented programming [Sch+10] where it is utilized to amend object-oriented programs when integrating features into a base system.

Allen and Garlan [AG97] applied the role concept in architectural modeling. They proposed the architecture description language WRIGTH which treats connectors between system components as a first-class concept. A connector is defined in terms of a set of roles and the so-called glue that defines the interaction between the roles. The interface between a component and a connector are a set of ports that can be filled by roles. Conceptually, a role defines the expected behavior or obligations of a component, or more specifically, a component port, in an interaction. As such, they are not used as an abstraction for context-dependent adaptations. The expected behavior of a component for filling a certain role and the connector glue is formally defined in terms of a CSP-style process algebra. The compatibility of a role and a port is formalized by a refinement relation. Since the description language is based on CSP, both the conformance and the absence of deadlocks within connector protocols can be checked using standard tools, including CSP model checkers.

In the Helena approach by Hennicker and Klarl [HK14], roles are played by components within *ensembles* to collaborate towards fulfilling a common goal. Components can play multiple roles in different ensembles simultaneously. Furthermore, the approach includes explicit relationships in the form of *role connectors* which define the messages that can be sent between roles. Components serve as storage for data and provide operations that can be invoked by their respective roles. However, a component does not exhibit any active behavior and cannot communicate with another component without the use of its roles. For that reason, roles in Helena do not and cannot adapt the behavior of their player. Ensembles which are specified in HelenaLight [Kla15], a simplified variant of the approach, can be formally verified using the SPIN model checker. For that an ensemble specification is translated into Promela, the input language of SPIN. Role behavior is given in terms of process expressions. The limited interaction between roles and components in Helena can be easily modeled in the RBA approach using synchronization over actions representing the component's operations.

Roles are a prominent modeling concept for designing and implementing *multi-agent systems* [Cab+10]. Here, a role is an abstraction of the expected behavior of an agent or a set of agents and defined in terms of capabilities, obligations, and requirements. Role-based approaches for multi-agent systems are not compositional in the sense that the combination of a role with an agent adapts the agent's behavior. Rather, it is assumed that an agent acquiring a certain role already exhibits the necessary behavior. Some

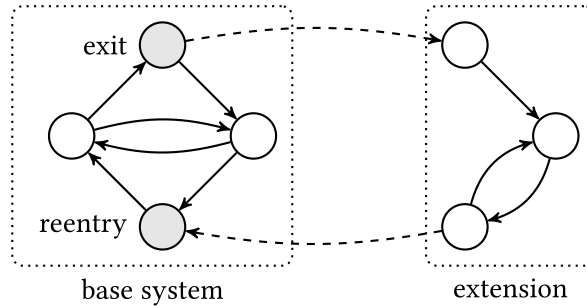


Figure 8.12: Sequential composition of a base system with an extension

approaches propose a refinement of the abstract role behavior to create the concrete agent behavior, e.g. [RS02].

Several model-checking approaches for *role-based access control* (RBAC) policies have been presented in the literature, e.g., [AT03; RB09; ZRG05]. In this context, roles are mainly associated with rights and exhibit no behavior themselves. In particular, roles are usually assigned to users and do not adapt the behavior of system components. The analysis of policies is mainly focused on whether it is possible for a user to gain access to a resource for a given set of roles and if so, which sequence of steps it takes.

8.3.2 Aspect-oriented approaches

Fisler and Krishnamurthi presented a methodology for the modular verification of collaboration-based software [FK01]. It relies on a quasi-sequential composition of extensions with a base system where both the base system and the extensions may arise from the parallel composition of several transition systems. Each base system provides an extension interface comprising an exit state and a reentry state that redirect the control flow into an extension and serve as a return point, respectively. The composition of a base system with an extension takes the union of the base system's and the extension's state spaces and connects the exit state with the initial state of the extension as well as the final state of the extension with the reentry state, as shown in Figure 8.12. Extensions may only add states and transitions, but cannot remove or modify existing states and transitions in the base system. The modular model checking of a collaboration proceeds in three steps. First, CTL model checking is applied to the base system only and the subformulas that are satisfied in the interface states are stored. Second, the extension is checked under the assumption that the subformulas that hold in the exit and reentry states of the base system are also satisfied in the initial and final states of the extension. Finally, it is checked whether the labeling of the extension's final state has been preserved by the CTL model checking procedure. If so, it can be concluded that the combination of the base system with the extension also satisfies the considered CTL formula. A similar approach has been applied for the verification of aspects that are inlined into the control

flow of a program [KFG04]. The approach has later been extended by Thang et al. to allow for a more flexible modeling [TK03b]. In the improved approach a base system may have multiple exit and reentry states. Furthermore, the transitions redirecting the control flow into the extension may be prioritized. This allows for an adaptation of the base system's behavior by suppressing certain transitions and replacing them with transitions of the extension. Even though the RBA formalism utilizes a parallel composition of role and player behavior, a sequential composition is achievable as well. Redirecting the control flow into the role is easily accomplished using overriding transitions. Once the control flow has reached the role, all actions of the player are blocked such that only the role behavior is executed.

In [Sip03], the formalization of aspects for modular transition systems is considered. Modular transition systems are specified using a first-order representation of transitions over the variables comprising the state space, similar to a description using guarded commands. Aspects, i.e., modularized modifications of a base system, are defined by a set of additional variables and a set of modification instructions. There are two kinds of modifications. An *abstraction* projects out one or more variables resulting in a weakening of guards and the removal of variable updates, effectively adding new behavior. Abstraction may be followed with the second kind of modification, *restriction*. A restriction strengthens guards and possibly adds variable updates. Therefore, it removes behavior from the system. The combination of abstraction and restriction allows for a modification of the base system's behavior when applying an aspect. The formalization is general enough to cover most of the constructs offered by AspectJ [Kic+01], an aspect-oriented extension of the Java programming language. The approach has a strong resemblance to superimposition, so the discussion of the relation to the RBA formalism given in Section 8.3.1 applies here as well.

8.3.3 Feature-oriented approaches

A notable similarity between features and roles is the effect they have if added to a base system and acquired by a player, respectively. In both instances the base behavior may be extended or modified. The similarities are even more profound in the case of dynamic features that can be added and removed during runtime of the system in order to adapt to changing requirements. A prominent formal model for feature-oriented systems tailored to an analysis using model checking are FTS [Cla+10]. The transitions of an FTS are annotated with *feature guards* that define in which feature combinations a transition is available. Thus, an FTS is a *family model* that represents a whole product line. The parallel composition operator on transition systems can be extended straightforwardly to FTS. However, the FTS formalism does not provide a composition operator for integrating an additional feature into an existing FTS. Instead, the superimposition construct has been applied to define FTS in a modular fashion [Cla+11; Cla+14]. For modeling adaptive systems and dynamic software product lines Cordy et al. introduced A-FTS [Cor+13a]. In

an A-FTS the feature guard is not only defined over the current feature combination, but also the feature combination in the successor state.

The role annotations in RBA are conceptually and structurally similar to the feature guards in FTS. This raises the question on how both formalisms compare w.r.t. their expressiveness. To answer this question, we define an embedding of FTS into RBA. For that, we first recall the definition of FTS. In the following, let $N = \{ f_0, f_1, \dots, f_n \}$ denote the set of features.

Definition 8.1 (Featured transition system [Cla+11])**.** An *FTS* is a tuple $fts = (S, Act, trans, d, \gamma, S^{init})$ where

- S is a finite set of states,
- Act is a set of actions,
- $trans \subseteq S \times Act \times S$ is a set of transitions,
- $d \subseteq \mathcal{P}(N)$ is a feature model,
- $\gamma\colon\ trans \to \big(\{0,1\}^{|N|} \to \{0,1\}\big)$ is a function labeling each transition with a feature expression, and
- $S^{init} \subseteq S$ is the set of initial states.

We now turn to the transformation of an FTS into an RBA. The main idea here is to treat an active feature as a role that is actively played. Since a feature guard is a Boolean expression over features, and thus represents multiple feature combinations symbolically, but role annotations describe role-playings explicitly, an FTS-transition may correspond to multiple RBA-transitions. We define the *corresponding role annotation of a feature combination* $p \in \mathcal{P}(N)$ as $X(p) = \big\{ f : f \in p \big\} \cup \big\{ \overline{f} : f \notin p \big\}$.

Definition 8.2 (RBA induced by an FTS)**.** The *corresponding RBA of an FTS* $fts = (S, Act, trans, d, \gamma, S^{init})$ is defined as

$$\mathcal{A}[fts] \;=\; \big(\, S,\ Act,\ \langle N, \varnothing \rangle,\ \longrightarrow,\ S^{init} \,\big)$$

where $\longrightarrow$ is the smallest transition relation fulfilling the following rule.

$$\frac{(s,\alpha,s') \in trans \qquad \gamma\big((s,\alpha,s')\big)\big([f_1 \in p], [f_2 \in p], \dots, [f_n \in p]\big) = 1 \qquad p \in \mathcal{P}(N)}{s \xrightarrow{\alpha/X(p)} Dirac(s')}$$

As it is the case with FTS, the resulting RBA represents the whole product line. Towards a family-based analysis, the *lifting* approach [PS08] can be applied to encode the feature combination into the state space of the model. In particular, the set of all valid feature combinations constitutes the state space of the role-playing coordinator.

Definition 8.3 (Coordinator arising from a feature model)**.** The *coordinator arising from the feature model* of an FTS $fts = (S, Act, trans, d, \gamma, S^{init})$ is defined as

$$\mathrm{C} \;=\; \big(\, d,\ Act,\ \langle N, \varnothing \rangle,\ \longrightarrow,\ d \,\big)$$

where $\longrightarrow = \{ (p, \alpha, X(p), p) : p \in d, \alpha \in Act \}$.

A coordinator state for a feature combination $p \in \mathcal{P}(N)$ has self-loops annotated with the corresponding role annotation for the feature combination p. Then, the composition of the coordinator with the RBA arising from the FTS ensures that only the roles associated with the selected feature combination are played. With that, the coordinator fulfills the same purpose as the *feature controller* in the work of Dubslaff et al. [DBK15]. Utilizing this representation of the feature combination, modeling a dynamic software product line amounts to simply adding transitions between the states of the coordinator such that the configuration can be changed during the runtime of the system. A similar approach has also been applied in delta-oriented programming where the construct was named *reconfiguration automaton* [DS11]. In summary, FTS can be fully embedded into RBA.

9 Conclusion

The goal of this thesis was to adopt the concepts of features and roles that have been introduced to design and implement variability-intensive as well as adaptive systems in the context of formal modeling and analysis. In particular, the focus was on a compositional definition of system models or model families and on a quantitative analysis by means of probabilistic model checking. The proposed formal approaches are complemented by suitable modeling languages and accompanying tooling, aiming to support the software development process.

For modeling and analyzing *feature-oriented systems*, the tool ProFeat has been extended in various directions. Support for feature attributes has been added which allow a more compact definition of product lines with numerical parameters. Furthermore, the implementation has been extended with the option of a one-by-one analysis where each instance of a system family is analyzed individually. The appropriate analysis method, i.e., either an all-in-one or a one-by-one analysis, can be chosen freely without requiring any modification of the model definition. An empirical evaluation comparing the analysis approaches revealed that none of them is clearly superior in all cases. For the considered systems, an all-in-one analysis was significantly faster for system families comprising numerous variants sharing a lot of common behavior. The one-by-one analysis was superior for systems defined in terms of parameters and with little sharing between instances. As a result of the translational approach of ProFeat, analysis results are ultimately provided by the utilized model checker and refer to the translated model. For an improved comprehensibility, a post-processing step has been added that transforms the results such that they correspond to the original ProFeat model. Moreover, ProFeat can generate a symbolic representation of the analysis results in terms of a propositional formula or an MTBDD. In case study involving a body sensor network product line, the MTBDD-representation proved to be useful for drawing high-level conclusions about the influence of certain features on the reliability of the system. Further case studies have shown the practical applicability of ProFeat for the analysis of product lines and other system families. Moreover, the usefulness of the feature concept for defining single systems has been shown where the adaptivity of the system was captured using dynamic features.

Role-based automata have been introduced to define the operational behavior of *role-based systems*. The compositional modeling approach provides an operator for role-binding which formalizes the interplay between a role and its player. The coordination of role-playing is captured by an automata-based component, the role-playing coordinator.

The semantics of an RBA representing the behavior of a role-based system under a coordinator is given in terms of an MDP. For describing systems involving roles succinctly, a role-oriented modeling language has been introduced. This language can be seen as an extension of PRISM's input language extended with various constructs for defining role-specific behavior. The implementation leverages PRISM for the quantitative analysis by translating a role-based model into the input language of PRISM. The translation encodes role-playing annotations into the action labels within the translated PRISM model which makes role-playing explicit. In case a system violates its specification, this explicit encoding of the involved roles facilitates the detection of interactions in counterexample traces. Furthermore, since roles belong to specific compartments that provide the context in which roles are played, the detection of interactions between different contexts is possible. The latter has been demonstrated in the analysis of a peer-to-peer file transfer protocol where interactions only occurred in the combination of several networks, but not within isolated networks. A second experimental study demonstrated the potential of detecting interactions that only influenced the efficiency of the system, but did not compromise its functional correctness. For the declarative and compositional definition of role-playing coordinators, an approach utilizing the channel-based exogenous coordination language Reo has been presented. The quantitative analysis of systems consisting of stochastic modules coordinated by a Reo connector is enabled by a prototype implementation which translates a textual description of a Reo circuit into an extended version of the PRISM language. This implementation is also applicable for analyzing role-based systems with exogenous coordination, as has been demonstrated by a second experimental study of the file-transfer protocol.

Future work. The work on feature-oriented systems presented in this thesis can be extended in several directions. The ProFeat language allows decomposing the behavior of a feature-oriented system into feature modules. However, within feature modules the definition of feature-specific behavior follows the annotative approach, i.e., commands are explicitly guarded by specific feature combinations. Thus, cross-cutting features cannot be cleanly modularized. To overcome this limitation, a superimposition construct [Kat93] could be integrated into the ProFeat language. Using the framework presented in [Dub19], the implementation can be extended accordingly to support both parallel composition and superimposition.

Another possible extension of the translation is the integration of an automated symmetry reduction for multi-features. Since instances of a multi-feature are created from a common template, they are potentially symmetric which allows replacing them with a generic representative [DM06]. For this it has to be investigated under which conditions the reduction may be safely applied.

Conceptually, it is possible to combine the all-in-one and one-by-one analysis approaches. For that the set of feature combinations is partitioned into a set of "sub-families".

The sub-families are analyzed one-by-one where the analysis of each sub-family utilizes the all-in-one approach. Von Rhein [Rhe16] reported that the combined approach uses less time than the one-by-one approach and less memory than the all-in-one approach for selected (non-probabilistic) models. ProFeat currently has rudimentary support for this combined analysis. It could be investigated whether the results reported by von Rhein can also be replicated for stochastic system families and whether a heuristic for the optimal number of partitions can be derived.

The symbolic result representations provided by ProFeat are only applicable for static product lines. In the future, theory and an implementation for meaningful explanations for dynamic feature-oriented systems could be developed. The approach may benefit from an explicit encoding of active feature behavior, similar to the role-playing annotations in RBA which make role interactions explicit.

A possible extension of the work on role-based systems is to consider a transfer of the concepts to timed automata or probabilistic timed automata for reasoning about real-time properties. Another direction for future work is the investigation under which conditions and restrictions a compositional reasoning or assume-guarantee reasoning is possible for RBA. Furthermore, the formalism may be combined with an explicit context model. Currently, contexts and context changes are only implicitly specified as part of the role-playing coordinator. An explicit modeling of contexts enables reasoning about context-changes and their impact on quantitative properties of the system.

Several extensions of the implementation are possible. Encoding of role-playing into the state space of the translated model is already possible, but the counterpart on the logic side is missing. The property specification language of PRISM could be extended with operators for reasoning about role-playing. An automated translation into the standard property language could be added by applying the techniques described in [DV90].

List of Figures

3.1 Feature diagram for a simple car product line 17
3.2 The feature-oriented development process 18
3.3 Producer-consumer running example 21
3.4 Feature diagram of the producer-consumer system 21
3.12 ProFeat workflow overview 30
3.13 A feature module and its translation 33
3.16 Translation of synchronization with controller 35
3.19 MTBDD representations of analysis results 39

4.1 MTBDD sizes of producer-consumer models 43
4.2 Analysis time of producer-consumer model (buffer size) 44
4.3 Analysis time of producer-consumer model (workers) 44
4.4 Analysis time of producer-consumer model (distributions) 45
4.5 Feature diagram for the body sensor network product line 48
4.8 MTBDD representation of BSN analysis results 50
4.9 Network system with cube topology 53
4.12 Probability of mapping failure in network model 55
4.13 Minimal required energy in network model 56
4.16 Probability of successful adaptation 59

5.1 CROM graphical notation 69
5.2 Role model for the banking example 71
5.3 RBA for an Account instance 73
5.4 RBA for Target and Checking roles 73
5.5 Rules for the parallel composition of RBA 74
5.6 Rules for role-binding in RBA 75
5.7 Results of binding Checking and Target roles 76
5.8 Example for non-commuting role-binding 80
5.9 Example showing different orders of role-binding 81
5.10 A role-playing coordinator for the banking example 82
5.11 Rules for the composition of an RBA and a coordinator 83
5.12 MDP for banking system 84

6.1 Overview for analysis of role-based systems 85
6.2 Type definitions for the banking scenario 87

6.3 Instantiation of a role-based system within the banking scenario 88
6.6 Decision tree for extending an incomplete system 91
6.8 Banking systems satisfying a set of constraints 93
6.13 A module implementing a component and the corresponding RBA 96
6.15 A module implementing a role component and the corresponding RBA . 99
6.17 Coordinator impementation and RBA for the banking example 101
6.23 Approaches for the composition of RBA 106
6.24 Rules for the parallel composition of maMDPs 108
6.25 maMDPs for Account natural and Checking role 110
6.26 Resulting maMDP for role-binding 110
6.27 Rules for the composition of an maMDP with a coordinator maMDP . . . 114
6.28 Rules for the parallel composition of PRISM modules 116
6.29 Composition of modules with multi-actions 117
6.35 Translation for encoding the visible action α into the state space 123
6.36 Translation for encoding role-playing of roles s and t into the state space 124

7.1 ReoCompiler overview 128
7.2 Reo node types 130
7.3 Alternator in Reo 130
7.4 Constraint automaton for a FIFO1 channel 131
7.5 Rules for the product of constraint automata 132
7.6 Rules for role-binding in CA 134
7.7 Role-binding patterns 134

8.5 Overlapping networks with a shared station 144
8.7 Relative analysis times for role-based models 147
8.8 An automated production cell with three robots 148
8.9 Maximal probability of fully processing n workpieces 149
8.10 Two automated production cells sharing a robot 150
8.11 A station component and its bound role components 152
8.12 Sequential composition of a base system with an extension 161

List of Tables

4.1 Sizes of parametrized models 46
4.2 Analysis for paramtrized benchmark models 47
4.3 Sizes of elevator models 51
4.4 Analysis times for elevator product lines 52

5.1 Ontological characterization of meta-types 68

8.1 Model sizes and analysis times for file-transfer model 147
8.2 Expected throughput of production cells 150
8.3 Analysis results for the file-transfer model with 3 stations 153
8.4 Model sizes and analysis times for the file-transfer model with 3 stations 154

List of Listings

3.5	Excerpt of the feature model for the producer-consumer system	22
3.6	Declaration and implementation of the Worker feature	24
3.7	Feature controller for the producer-consumer system	25
3.8	Parametrization of the producer-consumer model	26
3.9	A FIFO buffer implementation parametrized over the capacity	27
3.10	Definition of a Worker's energy consumption	28
3.11	Property specifications utilizing ProFeat's language extensions	29
3.14	Feature controller of the producer-consumer system	34
3.15	Translated feature controller command	34
3.17	Simple feature-oriented model	37
3.18	ProFeat analysis output	38
4.6	Analysis results for BSN model	49
4.7	Post-processed analysis results for BSN model	49
4.10	Deactivation of links in the network system controller	54
4.11	Nondeterministic task mapping in the network system model	54
4.14	Feature module for asynchronous message passing	58
4.15	Command for sending message	58
6.4	Binding multiple roles using nested binding	88
6.5	Incomplete set of constraints for instantiating multiple systems	90
6.7	Completed set of constraints for banking example	92
6.9	Description of a banking system	93
6.10	Type-restricted quantifier	93
6.11	An implementation declaration linking a module and a component type	94
6.12	Shorthand notation for providing an implementation for a type	95
6.14	Application of indexed action labels	97
6.16	Module for an Account with internal actions	100
6.18	Definition of role priorities	102
6.19	Function definition examples	103
6.20	Coordinator for transactions	104
6.21	Quantification over action labels	105
6.22	Definition of rewards	106
6.30	Modules defining behavior of Account natural and Checking role	118

6.31 Translated modules for Account natural and Checking role 119
6.32 Generated module for preventing the synchronization of non-role actions 120
6.33 Coordinator specifying a temporal constraint 122
6.34 Translated role-playing coordinator 123
6.37 Source-level translation for encoding role-playing into the state space . . 125

8.1 Definition of a unidirectional ring network topology 140
8.2 Module implementing a station 141
8.3 Module implementing a server role 142
8.4 Coordinator for server roles 143
8.6 Action trace for a property violation 145

Acronyms

AST abstract syntax tree

BDD binary decision diagram

CA constraint automaton

CDNF canonical disjunctive normal form

CROI Compartment Role Object Instance

CROM Compartment Role Object Model

CTL computation-tree logic

CTMC continuous-time Markov chain

DSPL dynamic software product line

DTMC discrete-time Markov chain

FOL first-order logic

FTS featured transition system

LTL linear temporal logic

maMDP Markov decision process with multi-actions

MDP Markov decision process

MTBDD multi-terminal binary decision diagram

PCA probabilistic constraint automaton

PCTL probabilistic computation tree logic

RBA role-based automaton

RBAC role-based access control

SPCA simple probabilistic constraint automaton

SPL software product line

TVL Textual Variability Language

Bibliography

[Ach+09] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. "Modeling Context and Dynamic Adaptations with Feature Models". In: *4th International Workshop Models@run.time at Models 2009 (MRT'09).* 2009, p. 10.

[AD90] Rajeev Alur and David L. Dill. "Automata For Modeling Real-Time Systems". In: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings.* Ed. by Mike Paterson. Vol. 443. Lecture Notes in Computer Science. Springer, 1990, pp. 322–335. DOI: 10.1007/BFb0032042.

[AG97] Robert Allen and David Garlan. "A Formal Basis for Architectural Connection". In: *ACM Trans. Softw. Eng. Methodol.* 6.3 (1997), pp. 213–249. DOI: 10.1145/258077.258078.

[AH96] Rajeev Alur and Thomas A. Henzinger. "Reactive Modules". In: *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996.* 1996, pp. 207–218. DOI: 10.1109/LICS.1996.561320.

[AHK03] Suzana Andova, Holger Hermanns, and Joost-Pieter Katoen. "Discrete-Time Rewards Model-Checked". In: *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers.* Ed. by Kim Guldstrand Larsen and Peter Niebert. Vol. 2791. Lecture Notes in Computer Science. Springer, 2003, pp. 88–104. DOI: 10.1007/978-3-540-40903-8_8.

[AK09] Sven Apel and Christian Kästner. "An Overview of Feature-Oriented Software Development". In: *Journal of Object Technology* 8.5 (2009), pp. 49–84.

[Ake78] Sheldon B. Akers. "Binary Decision Diagrams". In: *IEEE Transactions Computers* 27.6 (June 1978), pp. 509–516. ISSN: 0018-9340. DOI: 10.1109/TC.1978.1675141.

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. "FEATUREHOUSE: Language-independent, automated software composition". In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings.* IEEE, 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.

[Ape+11] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. "Detection of Feature Interactions Using Feature-aware Verification". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 372–375. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100075.

[Ape+13a] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. "Feature-interaction detection based on feature-based specifications". In: *Comput. Networks* 57.12 (2013), pp. 2399–2409. DOI: 10.1016/j.comnet.2013.02.025.

[Ape+13b] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. "Strategies for Product-line Verification: Case Studies and Experiments". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 482–491. ISBN: 978-1-4673-3076-3.

[Ape07] Sven Apel. "The role of features and aspects in software development: similarities, differences, and synergetic potential". PhD thesis. Otto-von-Guericke University Magdeburg, Germany, 2007. ISBN: 978-3-8364-3344-0. URL: http://diglib.uni-magdeburg.de/Dissertationen/2007/sveapel.htm.

[Arb03] Farhad Arbab. "Abstract Behavior Types: A Foundation Model for Components and Their Composition". English. In: *Formal Methods for Components and Objects*. Ed. by FrankS. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Vol. 2852. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 33–70. ISBN: 978-3-540-20303-2. DOI: 10.1007/978-3-540-39656-7_2.

[Arb04] Farhad Arbab. "Reo: a channel-based coordination model for component composition". In: *Mathematical Structures in Computer Science* 14 (03 June 2004), pp. 329–366. ISSN: 1469-8072. DOI: 10.1017/S0960129504004153.

[Asi+11] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, and Alessandro Fantechi. "Formal Description of Variability in Product Families". In: *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*. Ed. by Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John, and Klaus Schmid. IEEE Computer Society, 2011, pp. 130–139. DOI: 10.1109/SPLC.2011.34.

[AT03] Tanvir Ahmed and Anand R. Tripathi. "Static verification of security requirements in role based CSCW systems". In: *Proceedings of the eighth ACM symposium on Access control models and technologies*. ACM. 2003, pp. 196–203.

[BA12] Fernando Sérgio Barbosa and Ademar Aguiar. “Modeling and Programming with Roles: Introducing JavaStage”. In: *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Eleventh SoMeT ’12, Genoa, Italy, September 26~28, 2012.* 2012, pp. 124–145. DOI: 10.3233/978-1-61499-125-0-124.

[BA95] Andrea Bianco and Luca de Alfaro. “Model checking of probabilistic and non-deterministic systems”. In: *FSTTCS’95.* Vol. 1026. LNCS. 1995, pp. 499–513.

[Bab+05] Özalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, and Aad P. A. van Moorsel. In: *Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations [the book is a result from a workshop at Bertinoro, Italy, Summer 2004].* Ed. by Özalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad P. A. van Moorsel, and Maarten van Steen. Vol. 3460. Lecture Notes in Computer Science. Springer, 2005, pp. 1–20. DOI: 10.1007/11428589_1.

[Bah+93] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. “Algebraic decision diagrams and their applications”. In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993.* Ed. by Michael R. Lightner and Jochen A. G. Jess. IEEE, 1993, pp. 188–191. DOI: 10.1109/ICCAD.1993.580054.

[Bai+06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. “Modeling component connectors in Reo by constraint automata”. In: *Science of Computer Programming* 61.2 (2006). Second International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA’03), pp. 75–113. ISSN: 0167-6423. DOI: 10.1016/j.scico.2005.10.008.

[Bai+10] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. “Design and Verification of Systems with Exogenous Coordination Using Vereofy”. English. In: *Leveraging Applications of Formal Methods, Verification, and Validation.* Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 6416. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 97–111. ISBN: 978-3-642-16560-3. DOI: 10.1007/978-3-642-16561-0_15.

[Bai+13] Christel Baier, Benjamin Engel, Sascha Klüppelholz, Steffen Märcker, Hendrik Tews, and Marcus Völp. “A Probabilistic Quantitative Analysis of Probabilistic-Write/Copy-Select”. In: *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings.* Ed. by Guillaume Brat, Neha Rungta, and Arnaud Venet. Vol. 7871. Lecture

Notes in Computer Science. Springer, 2013, pp. 307–321. DOI: 10.1007/978-3-642-38088-4_21.

[Bai+14] Christel Baier, Marcus Daum, Clemens Dubslaff, Joachim Klein, and Sascha Klüppelholz. "Energy-Utility Quantiles". In: *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings.* Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 8430. Lecture Notes in Computer Science. Springer, 2014, pp. 285–299. DOI: 10.1007/978-3-319-06200-6_24.

[Bai+18] Christel Baier, Philipp Chrszon, Clemens Dubslaff, Joachim Klein, and Sascha Klüppelholz. "Energy-Utility Analysis of Probabilistic Systems with Exogenous Coordination". In: *It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab.* 2018, pp. 38–56. DOI: 10.1007/978-3-319-90089-6_3.

[Bai05] Christel Baier. "Probabilistic Models for Reo Connector Circuits". In: *J. UCS* 11.10 (2005), pp. 1718–1748. DOI: 10.3217/jucs-011-10-1718.

[Bat04] Don S. Batory. "Feature-Oriented Programming and the AHEAD Tool Suite". In: *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom.* Ed. by Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum. IEEE Computer Society, 2004, pp. 702–703. DOI: 10.1109/ICSE.2004.1317496.

[Bat05] Don Batory. "Feature Models, Grammars, and Propositional Formulas". English. In: *Software Product Lines.* Ed. by Henk Obbink and Klaus Pohl. Vol. 3714. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 7–20. ISBN: 978-3-540-28936-4. DOI: 10.1007/11554844_3.

[Bäu+98] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. "The role object pattern". In: *Washington University Dept. of Computer Science.* Citeseer. 1998.

[BBT06] Matteo Baldoni, Guido Boella, and Leendert W. N. van der Torre. "Roles as a Coordination Construct: Introducing powerJava". In: *Electron. Notes Theor. Comput. Sci.* 150.1 (2006), pp. 9–29. DOI: 10.1016/j.entcs.2005.12.021.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5.

[BD77] Charles W. Bachman and Manilal Daya. "The Role Concept in Data Models". In: *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3.* VLDB '77. Tokyo, Japan: VLDB Endowment, 1977, pp. 464–476. URL: http://dl.acm.org/citation.cfm?id=1286580.1286629.

[Bei90] Boris Beizer. *Software testing techniques (2. ed.)* Van Nostrand Reinhold, 1990. ISBN: 978-0-442-20672-7.

[Ben+95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems". In: *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*. Ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Vol. 1066. Lecture Notes in Computer Science. Springer, 1995, pp. 232–243. DOI: 10.1007/BFb0020949.

[Bey+07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The software model checker Blast". In: *International Journal on Software Tools for Technology Transfer* 9.5-6 (2007), pp. 505–525. DOI: 10.1007/s10009-007-0044-z.

[BG10] Stephanie Balzer and Thomas R. Gross. "Modular reasoning about invariants over shared state with interposed data members". In: *Proceedings of the 4th ACM Workshop Programming Languages meets Program Verification, PLPV 2010, Madrid, Spain, January 19, 2010.* 2010, pp. 49–56. DOI: 10.1145/1707790.1707794.

[BGE07] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. "A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships". English. In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 323–346. ISBN: 978-3-540-73588-5. DOI: 10.1007/978-3-540-73589-2_16.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings.* Ed. by Rance Cleaveland. Vol. 1579. Lecture Notes in Computer Science. Springer, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. The MIT press, 2008.

[BK11] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in

Computer Science. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[BK98] Christel Baier and Martha Kwiatkoswka. “Model Checking for a Probabilistic Branching Time Logic with Fairness”. In: *Distributed Computing* 11.3 (1998), pp. 125–155.

[BMS12] Maurice H. ter Beek, Franco Mazzanti, and Aldi Sulova. “VMC: A Tool for Product Variability Analysis”. In: *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings.* 2012, pp. 450–454. DOI: 10.1007/978-3-642-32759-9_36.

[BO92] Don S. Batory and Sean W. O’Malley. “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. In: *ACM Trans. Softw. Eng. Methodol.* 1.4 (1992), pp. 355–398. DOI: 10.1145/136586.136587.

[Boc+04] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. “Virtual machine support for dynamic join points”. In: *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004.* Ed. by Gail C. Murphy and Karl J. Lieberherr. ACM, 2004, pp. 83–92. DOI: 10.1145/976270.976282.

[Boc+12] Christoph Bockisch, Andreas Sewe, Haihan Yin, Mira Mezini, and Mehmet Aksit. “An In-Depth Look at ALIA4J”. In: *Journal of Object Technology* 11.1 (2012), pp. 1–28. DOI: 10.5381/jot.2012.11.1.a7.

[Bry86] Randal E. Bryant. “Graph-based algorithms for boolean function manipulation”. In: *IEEE Transactions on Computers* 100.8 (1986), pp. 677–691.

[BTR05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. “Automated reasoning on feature models”. In: *Advanced Information Systems Engineering.* Springer. 2005, pp. 491–503.

[Bur+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. “Symbolic Model Checking: 10^{20} States and Beyond”. In: *Inf. Comput.* 98.2 (1992), pp. 142–170. DOI: 10.1016/0890-5401(92)90017-A.

[Cab+10] Giacomo Cabri, Letizia Leonardi, Luca Ferrari, and Franco Zambonelli. “Role-based software agent interaction models: a survey”. In: *The Knowledge Engineering Review* 25.04 (2010), pp. 397–419.

[Cal+03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. “Feature interaction: a critical review and considered forecast”. In: *Comput. Networks* 41.1 (2003), pp. 115–141. DOI: 10.1016/S1389-1286(02)00352-3.

[CBH11] Andreas Classen, Quentin Boucher, and Patrick Heymans. “A text-based approach to feature modelling: Syntax and semantics of TVL”. In: *Science of Computer Programming* 76.12 (2011), pp. 1130–1143.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. ISBN: 978-0-201-30977-5.

[CE81] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71. DOI: 10.1007/BFb0025774.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. ISBN: 978-0-262-03270-4. URL: http://books.google.de/books?id=Nmc4wEaLXFEC.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. "Formalizing cardinality-based feature models and their specialization". In: *Software Process: Improvement and Practice* 10.1 (2005), pp. 7–29.

[Chr+16a] Philipp Chrszon, Clemens Dubslaff, Christel Baier, Joachim Klein, and Sascha Klüppelholz. "Modeling Role-Based Systems with Exogenous Coordination". In: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. 2016, pp. 122–139. DOI: 10.1007/978-3-319-30734-3_10.

[Chr+16b] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. "Family-Based Modeling and Analysis for Probabilistic Systems - Featuring ProFeat". In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 287–304. DOI: 10.1007/978-3-662-49665-7_17.

[Chr+18] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. "ProFeat: feature-oriented engineering for family-based probabilistic model checking". In: *Formal Aspects of Computing* 30.1 (2018), pp. 45–75. DOI: 10.1007/s00165-017-0432-4.

[Chr+20] Philipp Chrszon, Christel Baier, Clemens Dubslaff, and Sascha Klüppelholz. "From features to roles". In: *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020*. ACM, 2020, 19:1–19:11. DOI: 10.1145/3382025.3414962.

[Chr14] Philipp Chrszon. "Quantitative Analyse von Produktlinien mit PRISM". MA thesis. Technische Universität Dresden, Germany, 2014.

[Cim+00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. "NUSMV: A New Symbolic Model Checker". In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425. DOI: 10.1007/s100090050046.

[Cla+10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. "Model checking lots of systems: efficient verification of temporal properties in software product lines". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010.* Ed. by Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel. ACM, 2010, pp. 335–344. DOI: 10.1145/1806799.1806850.

[Cla+11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. "Symbolic model checking of software product lines". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011.* Ed. by Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic. ACM, 2011, pp. 321–330. DOI: 10.1145/1985793.1985838.

[Cla+12] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. "Model checking software product lines with SNIP". In: *International Journal on Software Tools for Technology Transfer* 14.5 (2012), pp. 589–612. ISSN: 1433-2787. DOI: 10.1007/s10009-012-0234-1.

[Cla+13] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking". In: *Software Engineering, IEEE Transactions on* 39.8 (Aug. 2013), pp. 1069–1089. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.86.

[Cla+14] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. "Formal semantics, modular specification, and symbolic verification of product-line behaviour". In: *Science of Computer Programming* 80 (2014), pp. 416–439.

[Cla+93] E. M. Clarke, M. Fujita, P. C. McGeers, K. L. McMillan, J. C.-Y. Yang, and X.-J. Zhao. "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation". In: *Proc. International Workshop on Logic & Synthesis.* 1993.

[Cla+96] Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. "Exploiting Symmetry in Temporal Logic Model Checking". In: *Formal Methods Syst. Des.* 9.1/2 (1996), pp. 77–104. DOI: 10.1007/BF00625969.

[CN02] P.C. Clements and L.M. Northrop. *Software Product Lines: Practices and Patterns.* The SEI series in software engineering. Addison Wesley Professional, 2002. ISBN: 9780201703320.

[Cor+13a] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. "Model Checking Adaptive Software with Featured Transition Systems". In: *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques.* 2013, pp. 1–29. DOI: 10.1007/978-3-642-36249-1_1.

[Cor+13b] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. "ProVeLines: A Product Line of Verifiers for Software Product Lines". In: *Proceedings of the 17th International Software Product Line Conference Co-located Workshops.* SPLC '13 Workshops. Tokyo, Japan: ACM, 2013, pp. 141–146. ISBN: 978-1-4503-2325-3. DOI: 10.1145/2499777.2499781.

[Cor+13c] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. "Beyond boolean product-line model checking: dealing with feature attributes and multi-features". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013.* Ed. by David Notkin, Betty H. C. Cheng, and Klaus Pohl. IEEE Computer Society, 2013, pp. 472–481. DOI: 10.1109/ICSE.2013.6606593.

[CR14] James O. Coplien and Trygve Reenskaug. "The DCI Paradigm: Taking Object Orientation into the Architecture World". In: *Agile Software Architecture.* Ed. by Muhammad Ali Babar, Alan W. Brown, and Ivan Mistrik. Morgan Kaufmann, 2014, pp. 25–62. ISBN: 978-0-12-407772-0. DOI: 10.1016/B978-0-12-407772-0.00002-2.

[DA18] Kasper Dokter and Farhad Arbab. "Treo: Textual Syntax for Reo Connectors". In: *Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design, MeTRiD@ETAPS 2018, Thessaloniki, Greece, 15th April 2018.* Ed. by Simon Bliudze and Saddek Bensalem. Vol. 272. EPTCS. 2018, pp. 121–135. DOI: 10.4204/EPTCS.272.10.

[DBK15] Clemens Dubslaff, Christel Baier, and Sascha Klüppelholz. "Probabilistic Model Checking for Feature-Oriented Systems". In: *LNCS Transactions on Aspect Oriented Software Development* 12 (2015), pp. 180–220. DOI: 10.1007/978-3-662-46734-3_5.

[Dim+15] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. "Family-Based Model Checking Without a Family-Based Model Checker". In: *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings.* 2015, pp. 282–299. DOI: 10.1007/978-3-319-23404-5_18.

[Din+10] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. "A Dynamic Software Product Line Approach Using Aspect Models at Runtime". In: *Proc. of the 1st Workshop on Composition and Variability.* 2010.

[DKB14] Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. "Probabilistic model checking for energy analysis in software product lines". In: *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014.* Ed. by Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld. ACM, 2014, pp. 169–180. DOI: 10.1145/2577080.2577095.

[DM06] Alastair F. Donaldson and Alice Miller. "Symmetry Reduction for Probabilistic Model Checking Using Generic Representatives". In: *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006.* Ed. by Susanne Graf and Wenhui Zhang. Vol. 4218. Lecture Notes in Computer Science. Springer, 2006, pp. 9–23. DOI: 10.1007/11901914_4.

[DPZ02] Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. "A Generic Role Model for Dynamic Objects". In: *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings.* Ed. by Anne Banks Pidduck, John Mylopoulos, Carson C. Woo, and M. Tamer Özsu. Vol. 2348. Lecture Notes in Computer Science. Springer, 2002, pp. 643–658. DOI: 10.1007/3-540-47961-9_44.

[DS11] Ferruccio Damiani and Ina Schaefer. "Dynamic Delta-oriented Programming". In: *Proceedings of the 15th International Software Product Line Conference, Volume 2.* SPLC '11. ACM, 2011, 34:1–34:8. ISBN: 978-1-4503-0789-5. DOI: 10.1145/2019136.2019175.

[Dub19] Clemens Dubslaff. "Compositional Feature-Oriented Systems". In: *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings.* Ed. by Peter Csaba Ölveczky and Gwen Salaün. Vol. 11724. Lecture Notes in Computer Science. Springer, 2019, pp. 162–180. DOI: 10.1007/978-3-030-30446-1_9.

[DV90] Rocco De Nicola and Frits W. Vaandrager. "Action versus State based Logics for Transition Systems". In: *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings.* 1990, pp. 407–419. DOI: 10.1007/3-540-53479-2_17.

[FGR94] Alessandro Fantechi, Stefania Gnesi, and Gioia Ristori. "Model Checking for Action-Based Logics". In: *Formal Methods in System Design* 4.2 (1994), pp. 187–203. DOI: 10.1007/BF01384084.

[FK01] Kathi Fisler and Shriram Krishnamurthi. “Modular verification of collaboration-based software designs”. In: *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001.* 2001, pp. 152–163.

[FK92] David Ferraiolo and Richard Kuhn. “Role-Based Access Control”. In: *In 15th NIST-NCSC National Computer Security Conference.* 1992, pp. 554–563.

[Gen07] Valerio Genovese. “A meta-model for roles: Introducing sessions”. In: *Proceedings of the 2nd Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies.* 2007, pp. 27–38.

[GH03] Hassan Gomaa and Mohamed Hussein. “Dynamic Software Reconfiguration in Software Product Families”. In: *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers.* Ed. by Frank van der Linden. Vol. 3014. Lecture Notes in Computer Science. Springer, 2003, pp. 435–444. DOI: 10.1007/978-3-540-24667-1_33.

[GØ02] Kasper B. Graversen and Kasper Østerbye. “Aspect modelling as role modelling”. In: *OOPSLA 2002 Workshop on TS4AOSD.* 2002.

[GØ03] Kasper B. Graversen and Kasper Østerbye. “Implementation of a role language for object-specific dynamic separation of concerns”. In: *AOSD03 Workshop on Software-engineering Properties of Languages for Aspect Technologies.* 2003.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Vol. 1032. Lecture Notes in Computer Science. Springer, 1996. ISBN: 3-540-60761-7. DOI: 10.1007/3-540-60761-7.

[GOR06] Matthias Güdemann, Frank Ortmeier, and Wolfgang Reif. “Formal Modeling and Verification of Systems with Self-x Properties”. In: *Autonomic and Trusted Computing, Third International Conference, ATC 2006, Wuhan, China, September 3-6, 2006, Proceedings.* 2006, pp. 38–47. DOI: 10.1007/11839569_4.

[Gri00] Martin L. Griss. “Implementing product-line features with component reuse”. In: *Software Reuse: Advances in Software Reusability.* Springer, 2000, pp. 137–152.

[GS13] Carlo Ghezzi and Amir Molzam Sharifloo. “Model-based verification of quantitative non-functional properties for software product lines”. In: *Information and Software Technology* 55.3 (2013). Special Issue on Software Reuse and Product Lines Special Issue on Software Reuse and Product Lines, pp. 508–524. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.07.017.

[Gui+04] Giancarlo Guizzardi, Gerd Wagner, Nicola Guarino, and Marten van Sinderen. "An Ontologically Well-Founded Profile for UML Conceptual Models". English. In: *Advanced Information Systems Engineering*. Ed. by Anne Persson and Janis Stirna. Vol. 3084. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 112–126. ISBN: 978-3-540-22151-7. DOI: 10.1007/978-3-540-25975-6_10.

[Gui05] Giancarlo Guizzardi. "Ontological foundations for structural conceptual models". PhD thesis. Enschede, 2005. URL: http://doc.utwente.nl/50826/.

[GW00] Nicola Guarino and Christopher Welty. "A Formal Ontology of Properties". English. In: *Knowledge Engineering and Knowledge Management Methods, Models, and Tools*. Ed. by Rose Dieng and Olivier Corby. Vol. 1937. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 97–112. ISBN: 978-3-540-41119-2. DOI: 10.1007/3-540-39967-4_8.

[GW12] Giancarlo Guizzardi and Gerd Wagner. "Conceptual simulation modeling with onto-UML". In: *Winter Simulation Conference, WSC '12, Berlin, Germany, December 9-12, 2012*. Ed. by Oliver Rose and Adelinde M. Uhrmacher. WSC, 2012, 5:1–5:15. DOI: 10.1109/WSC.2012.6465328.

[Hah+10] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. "PARAM: A model checker for parametric Markov models". In: *Computer Aided Verification*. Springer. 2010, pp. 660–664.

[Hah+14] Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. "iscasMc: A Web-Based Probabilistic Model Checker". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 312–317. DOI: 10.1007/978-3-319-06410-9_22.

[Hal05] Terry A. Halpin. "ORM 2". In: *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, GADA, MIOS+INTEROP, ORM, PhDS, SeBGIS, SWWS, and WOSE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings*. Vol. 3762. Lecture Notes in Computer Science. Springer, 2005, pp. 676–687. DOI: 10.1007/11575863_87.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. "Context-oriented Programming". In: *Journal of Object Technology* 7.3 (2008), pp. 125–151. DOI: 10.5381/jot.2008.7.3.a4.

[He+06] Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. “Rava: Designing a Java Extension with Dynamic Object Roles”. In: *13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2006), 27-30 March 2006, Potsdam, Germany*. 2006, pp. 453–459. DOI: 10.1109/ECBS.2006.57.

[Hen+20] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. “The Probabilistic Model Checker Storm”. In: *CoRR* abs/2002.07080 (2020). eprint: 2002.07080. URL: https://arxiv.org/abs/2002.07080.

[Her02] Stephan Herrmann. “Object Teams: Improving Modularity for Crosscutting Collaborations”. In: *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers*. 2002, pp. 248–264. DOI: 10.1007/3-540-36557-5_19.

[Her07] Stephan Herrmann. “A precise model for contextual roles: The programming language ObjectTeams/Java”. In: *Applied Ontology* 2.2 (2007), pp. 181–207.

[HH14] Arnd Hartmanns and Holger Hermanns. “The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 593–598. DOI: 10.1007/978-3-642-54862-8_51.

[HK14] Rolf Hennicker and Annabelle Klarl. “Foundations for Ensemble Modeling – The Helena Approach”. In: *Specification, Algebra, and Software*. Ed. by Shusaku Iida, José Meseguer, and Kazuhiro Ogata. Vol. 8373. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 359–381.

[Hol97] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521.

[HP00] Klaus Havelund and Thomas Pressburger. “Model Checking JAVA Programs using JAVA PathFinder”. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 366–381. DOI: 10.1007/s100090050043.

[HQR98] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. “You Assume, We Guarantee: Methodology and Case Studies”. In: *Computer Aided Verification, 10th International Conference, CAV ’98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. Ed. by Alan J. Hu and Moshe Y. Vardi.

Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 440–451. doi: 10.1007/BFb0028765.

[HU02] Stefan Hanenberg and Rainer Unland. “Roles and Aspects: Similarities, Differences, and Synergetic Potential”. In: *Object-Oriented. Information Systems, 8th International Conference, OOIS 2002, Montpellier, France, September 2-5, 2002, Proceedings.* 2002, pp. 507–520. doi: 10.1007/3-540-46102-7_53.

[JA12] Sung-Shik TQ Jongmans and Farhad Arbab. “Overview of Thirty Semantic Formalisms for Reo”. In: *Sci. Ann. Comp. Sci.* 22.1 (2012), pp. 201–251.

[Jör+12] Sven Jörges, Anna-Lena Lamprecht, Tiziana Margaria, Ina Schaefer, and Bernhard Steffen. “A constraint-based variability modeling framework”. In: *International Journal on Software Tools for Technology Transfer* 14.5 (2012), pp. 511–530. doi: 10.1007/s10009-012-0254-x.

[JSM97] He Jifeng, K. Seidel, and A. McIver. “Probabilistic models for the guarded command language”. In: *Science of Computer Programming* 28.2 (1997). Formal Specifications: Foundations, Methods, Tools and Applications, pp. 171–192. issn: 0167-6423.

[Kan+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study.* Tech. rep. Carnegie-Mellon University Software Engineering Institute, 1990.

[Kat+11] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. “The ins and outs of the probabilistic model checker MRMC”. In: *Performance Evaluation* 68.2 (2011), pp. 90–104. doi: 10.1016/j.peva.2010.04.001.

[Kat93] Shmuel Katz. “A superimposition control construct for distributed systems”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.2 (1993), pp. 337–356.

[Kel76] Robert M. Keller. “Formal Verification of Parallel Programs”. In: *Communications of the ACM* 19.7 (1976), pp. 371–384. doi: 10.1145/360248.360251.

[KFG04] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. “Verifying aspect advice modularly”. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004.* 2004, pp. 137–146. doi: 10.1145/1029894.1029916.

[Kic+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. “Getting started with ASPECTJ”. In: *Commun. ACM* 44.10 (2001), pp. 59–65. doi: 10.1145/383845.383858.

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-Oriented Programming”. In: *ECOOP’97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*. Ed. by Mehmet Aksit and Satoshi Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer, 1997, pp. 220–242. DOI: 10.1007/BFb0053381.

[Kla15] Annabelle Klarl. “From Helena Ensemble Specifications to Promela Verification Models”. In: *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*. 2015, pp. 39–45. DOI: 10.1007/978-3-319-23404-5_4.

[Kle+18] Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, and David Müller. “Advances in Probabilistic Model Checking with PRISM: Variable Reordering, Quantiles and Weak Deterministic Büchi Automata”. In: *International Journal on Software Tools for Technology Transfer* 20.2 (2018), pp. 179–194. DOI: 10.1007/s10009-017-0456-3.

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. “PRISM: Probabilistic symbolic model checker”. In: *Computer Performance Evaluation: Modelling Techniques and Tools*. Springer, 2002, pp. 200–204.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems”. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

[KNP12] M. Z. Kwiatkowska, G. Norman, and D. Parker. “The PRISM Benchmark Suite”. In: *Proc. Quantitative Evaluation of Systems (QEST’12)*. https://github.com/prismmodelchecker/prism-benchmarks/. IEEE, 2012, pp. 203–204.

[Kru+15] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. “A survey on engineering approaches for self-adaptive systems”. In: *Pervasive and Mobile Computing* 17, Part B (2015). 10 years of Pervasive Computing’ In Honor of Chatschik Bisdikian, pp. 184–206. ISSN: 1574-1192. DOI: http://dx.doi.org/10.1016/j.pmcj.2014.09.009.

[KST14] Matthias Kowal, Ina Schaefer, and Mirco Tribastone. “Family-Based Performance Analysis of Variant-Rich Software Systems”. English. In: *Fundamental Approaches to Software Engineering*. Ed. by Stefania Gnesi and Arend Rensink. Vol. 8411. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 94–108. ISBN: 978-3-642-54803-1. DOI: 10.1007/978-3-642-54804-8_7.

[KT10] Tetsuo Kamina and Tetsuo Tamai. "A Smooth Combination of Role-based Language and Context Activation". In: *FOAL 2010 Proceedings* (2010), pp. 15–24.

[Küh+14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. "A Metamodel Family for Role-Based Modeling and Programming Languages". In: *Software Language Engineering*. Springer, 2014, pp. 141–160.

[Küh+15] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. "A combined formal model for relational context-dependent roles". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*. 2015, pp. 113–124. DOI: 10.1145/2814251.2814255.

[Küh+16] Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. "FRaMED: full-fledge role modeling editor (tool demo)". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, 2016, pp. 132–136. URL: http://dl.acm.org/citation.cfm?id=2997371.

[Küh17] Thomas Kühn. "A Family of Role-Based Languages". PhD thesis. Dresden University of Technology, Germany, 2017. URL: https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-228027.

[Kul16] Vidyadhar G. Kulkarni. *Modeling and analysis of stochastic systems*. Crc Press, 2016.

[LA15] Max Leuthäuser and Uwe Aßmann. "Enabling View-based Programming with SCROLL: Using roles and dynamic dispatch for etablishing view-based programming". In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. ACM. 2015, pp. 25–33.

[Lee59] C. Y. Lee. "Representation of Switching Circuits by Binary-Decision Programs". In: *Bell System Technical Journal* 38.4 (1959), pp. 985–999. ISSN: 1538-7305. DOI: 10.1002/j.1538-7305.1959.tb01585.x.

[LH09] Mengchi Liu and Jie Hu. "Information Networking Model". English. In: *Conceptual Modeling - ER 2009*. Ed. by AlbertoH.F. Laender, Silvana Castano, Umeshwar Dayal, Fabio Casati, and Jos√©PalazzoM. de Oliveira. Vol. 5829. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 131–144. ISBN: 978-3-642-04839-5. DOI: 10.1007/978-3-642-04840-1_12.

[Lie+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. “An analysis of the variability in forty preprocessor-based software product lines”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010.* Ed. by Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel. ACM, 2010, pp. 105–114. DOI: 10.1145/1806799.1806819.

[LMT07] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Angelo Troina. “Parametric probabilistic transition systems for system design and analysis”. In: *Formal Aspects of Computing* 19.1 (2007), pp. 93–109. DOI: 10.1007/s00165-006-0015-2.

[LPT09] K. Lauenroth, K. Pohl, and S. Toehning. “Model Checking of Domain Artifacts in Product Line Engineering”. In: *Automated Software Engineering, 2009. ASE ’09. 24th IEEE/ACM International Conference on.* Nov. 2009, pp. 269–280. DOI: 10.1109/ASE.2009.16.

[LPY95] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. “Model-Checking for Real-Time Systems”. In: *Fundamentals of Computation Theory, 10th International Symposium, FCT ’95, Dresden, Germany, August 22-25, 1995, Proceedings.* Ed. by Horst Reichel. Vol. 965. Lecture Notes in Computer Science. Springer, 1995, pp. 62–88. DOI: 10.1007/3-540-60249-6_41.

[Mae87] Pattie Maes. “Concepts and Experiments in Computational Reflection”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’87), Orlando, Florida, USA, October 4-8, 1987, Proceedings.* Ed. by Norman K. Meyrowitz. ACM, 1987, pp. 147–155. DOI: 10.1145/38765.38821.

[McC56] Edward J. McCluskey. “Minimization of Boolean Functions*”. In: *Bell System Technical Journal* 35.6 (1956). DOI: 10.1002/j.1538-7305.1956.tb03835.x.

[McM93] Kenneth L. McMillan. *Symbolic model checking.* Kluwer, 1993. ISBN: 978-0-7923-9380-1. DOI: 10.1007/978-1-4615-3190-6.

[MKK12] Riichiro Mizoguchi, Kouji Kozaki, and Yoshinobu Kitamura. “Ontological analyses of roles”. In: *Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on.* IEEE. 2012, pp. 489–496.

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL.* Springer, 2014. ISBN: 978-3-319-10541-3. DOI: 10.1007/978-3-319-10542-0.

[OMG11] Object Management Group OMG. *Unified Modeling Language (UML): Superstructure Version 2.4.1.* http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/. Aug. 2011.

[PA98] George A. Papadopoulos and Farhad Arbab. “Coordination Models and Languages”. In: ed. by Marvin V. Zelkowitz. Vol. 46. Advances in Computers. Elsevier, 1998, pp. 329–400. DOI: 10.1016/S0065-2458(08)60208-9.

[Pel93] Doron A. Peled. “All from One, One for All: on Model Checking Using Representatives”. In: *Computer Aided Verification, 5th International Conference, CAV ’93, Elounda, Greece, June 28 - July 1, 1993, Proceedings.* Ed. by Costas Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 409–423. DOI: 10.1007/3-540-56922-7_34.

[Per90] Barbara Pernici. “Objects with Roles”. In: *SIGOIS Bull.* 11.2-3 (Mar. 1990), pp. 205–215. ISSN: 0894-0819. DOI: 10.1145/91478.91542.

[Plo04] Gordon D. Plotkin. “A structural approach to operational semantics”. In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 17–139.

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[Pnu84] Amir Pnueli. “In Transition From Global to Modular Temporal Reasoning about Programs”. In: *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984.* Ed. by Krzysztof R. Apt. Vol. 13. NATO ASI Series. Springer, 1984, pp. 123–144. DOI: 10.1007/978-3-642-82453-1_5.

[PR01] Malte Plath and Mark Ryan. “Feature integration using a feature construct”. In: *Science of Computer Programming* 41.1 (2001), pp. 53–84.

[PS08] Hendrik Post and Carsten Sinz. “Configuration Lifting: Verification meets Software Configuration”. In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L’Aquila, Italy.* IEEE Computer Society, 2008, pp. 347–350. DOI: 10.1109/ASE.2008.45.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley Series in Probability and Statistics. Wiley, 1994. ISBN: 978-0-47161977-2. DOI: 10.1002/9780470316887.

[QS82] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings.* Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351. DOI: 10.1007/3-540-11494-7_22.

[RB09] Hind Rakkay and Hanifa Boucheneb. "Security Analysis of Role Based Access Control Models Using Colored Petri Nets and CPNtools". In: *Transactions on Computational Science IV.* Ed. by MarinaL. Gavrilova, C.J.Kenneth Tan, and EdwardDavid Moreno. Vol. 5430. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 149–176. ISBN: 978-3-642-01003-3. DOI: 10.1007/978-3-642-01004-0_9.

[Reo] Reo. *The Reo Compiler.* https://github.com/ReoLanguage/Reo.

[Rhe+15] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. "Presence-Condition Simplification in Highly Configurable Systems". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1.* Ed. by Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum. IEEE Computer Society, 2015, pp. 178–188. DOI: 10.1109/ICSE.2015.39.

[Rhe16] Alexander von Rhein. "Analysis Strategies for Configurable Systems". PhD thesis. University of Passau, 2016.

[Rie+02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. "Extending feature diagrams with UML multiplicities". In: *Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA.* Vol. 50. 2002.

[Rod+15] Genaína Nunes Rodrigues, Vander Alves, Vinicius Nunes, André Lanna, Maxime Cordy, Pierre-Yves Schobbens, Amir Molzam Sharifloo, and Axel Legay. "Modeling and Verification for Probabilistic Properties in Software Product Lines". In: *16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, January 8-10, 2015.* IEEE Computer Society, 2015, pp. 173–180. DOI: 10.1109/HASE.2015.34.

[RS02] Mark Ryan and Pierre-Yves Schobbens. "Agents and Roles: Refinement in Alternating-Time Temporal Logic". In: *Intelligent Agents VIII.* Ed. by John-JulesCh. Meyer and Milind Tambe. Vol. 2333. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 100–115. ISBN: 978-3-540-43858-8. DOI: 10.1007/3-540-45448-9_8.

[Rud93] Richard Rudell. "Dynamic variable ordering for ordered binary decision diagrams". In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93).* 1993, pp. 42–47. DOI: 10.1109/ICCAD.1993.580029.

[Sch+07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. "Generic semantics of feature diagrams". In: *Comput. Networks* 51.2 (2007), pp. 456–479. DOI: 10.1016/j.comnet.2006.08.008.

[Sch+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings.* Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 77–91. DOI: 10.1007/978-3-642-15579-6_6.

[SCT03] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. "A Selective, Just-in-Time Aspect Weaver". In: *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings.* Ed. by Frank Pfenning and Yannis Smaragdakis. Vol. 2830. Lecture Notes in Computer Science. Springer, 2003, pp. 189–208. DOI: 10.1007/978-3-540-39815-8_12.

[Seg08] Sergio Segura. "Automated Analysis of Feature Models Using Atomic Sets". In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops).* Ed. by Steffen Thiel and Klaus Pohl. Lero Int. Science Centre, University of Limerick, Ireland, 2008, pp. 201–207.

[Sie+12] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. "Predicting performance via automated feature-interaction detection". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland.* Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, pp. 167–177. DOI: 10.1109/ICSE.2012.6227196.

[Sip03] Henny B. Sipma. "A formal model for cross-cutting modular transition systems". In: *Proc. of Foundations of Aspect Languages Workshop (FOAL03).* 2003.

[SPL] SPLC. *Product Line Hall of Fame.* https://splc.net/fame.html.

[Ste00] Friedrich Steimann. "On the representation of roles in object-oriented and conceptual modelling". In: *Data & Knowledge Engineering* 35.1 (2000), pp. 83–106.

[Thü+14a] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A Classification and Survey of Analysis Strategies for Software Product Lines". In: *ACM Comput. Surv.* 47.1 (June 2014), 6:1–6:45. ISSN: 0360-0300. DOI: 10.1145/2580950.

[Thü+14b] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. “FeatureIDE: An extensible framework for feature-oriented software development”. In: *Sci. Comput. Program.* 79 (2014), pp. 70–85. DOI: 10.1016/j.scico.2012.06.002.

[TK03a] Nguyen Truong Thang and Takuya Katayama. “Dynamic Behavior and Protocol Models for Incremental Changes among a Set of Collaborative Objects”. In: *6th International Workshop on Principles of Software Evolution (IWPSE 2003), 1-2 September 2003, Helsinki, Finland.* 2003, pp. 45–50.

[TK03b] Nguyen Truong Thang and Takuya Katayama. “Towards a Sound Modular Model Checking of Collaboration-Based Software Designs”. In: *10th Asia-Pacific Software Engineering Conference (APSEC 2003), 10-12 December 2003, Chiang Mai, Thailand.* 2003, pp. 88–97.

[TUI05] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. “An adaptive object model with dynamic role binding”. In: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA.* 2005, pp. 166–175. DOI: 10.1145/1062455.1062498.

[Val92] Antti Valmari. “A Stubborn Attack on State Explosion”. In: *Formal Methods Syst. Des.* 1.4 (1992), pp. 297–322. DOI: 10.1007/BF00709154.

[Van+18] Andrea Vandin, Maurice H. ter Beek, Axel Legay, and Alberto Lluch-Lafuente. “QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems”. In: *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings.* 2018, pp. 329–337. DOI: 10.1007/978-3-319-95582-7_19.

[Var85] Moshe Y. Vardi. “Automatic Verification of Probabilistic Concurrent Finite-State Programs”. In: *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985.* IEEE Computer Society, 1985, pp. 327–338. DOI: 10.1109/SFCS.1985.12.

[VK02] Arie Van Deursen and Paul Klint. “Domain-specific language design requires feature descriptions”. In: *CIT. Journal of Computing and Information Technology* 10.1 (2002), pp. 1–17.

[VL16] Valentino Vranic and Milan Laslop. “Aspects and roles in software modeling: A composition based comparison”. In: *Comput. Sci. Inf. Syst.* 13.1 (2016), pp. 199–216. DOI: 10.2298/CSIS151207065V.

[Wei+17] Martin Weißbach, Philipp Chrszon, Thomas Springer, and Alexander Schill. "Decentrally Coordinated Execution of Adaptations in Distributed Self-Adaptive Software Systems". In: *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18-22*. 2017, pp. 111–120. DOI: 10.1109/SASO.2017.20.

[Wei08] David M. Weiss. "The Product Line Hall of Fame". In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IEEE Computer Society, 2008, p. 395. DOI: 10.1109/SPLC.2008.56.

[Wil+00] Poul Frederick Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. "Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 124–138. DOI: 10.1007/10722167_13.

[Zav03] Pamela Zave. "An experiment in feature engineering". In: *Programming methodology*. Springer, 2003, pp. 353–377.

[ZRG05] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. "Evaluating Access Control Policies Through Model Checking". In: *Information Security*. Ed. by Jianying Zhou, Javier Lopez, RobertH. Deng, and Feng Bao. Vol. 3650. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 446–460. ISBN: 978-3-540-29001-8. DOI: 10.1007/11556992_32.

[ZZ08] Haibin Zhu and MengChu Zhou. "Roles in Information Systems: A Survey". In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 38.3 (May 2008), pp. 377–396. ISSN: 1094-6977. DOI: 10.1109/TSMCC.2008.919168.