Bringing Model Checking Closer To Practical Software Engineering

Daniela Remenska

Dutch title:	Modelchecken dichter bij de praktijk van software engineering brengen
Printed by:	Uitgeverij BOXpress ‖ proefschriftmaken.nl
ISBN:	978-94-6295-432-8
Cover:	artwork by Daniela Remenska

A digital version is available at: www.ubvu.vu.nl/dissertations

This work is part of the research programme of the Foundation for Fundamental Research on Matter (FOM), which is part of the Netherlands Organisation for Scientific Research (NWO).

VRIJE UNIVERSITEIT

Bringing Model Checking Closer To Practical Software Engineering

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op maandag 8 februari 2016 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Daniela Remenska

geboren te Skopje, Macedonië

promotor:	prof.dr.ir. H.E. Bal
copromotors:	dr. J.A. Templon
	dr.ir. T.A.C. Willemse

Acknowledgements

They all said it's the most difficult one to write, but I did not believe them. After all, it's all about the people that made it happen. How hard can it be to express your gratitude? And then the day came when I kept staring at a blank page, while reflecting on the years that went by. Several of these numb evenings, it was never a good timing.

It all began with a "Welcome aboard!" and a big grin on your face, remember Jeff? Even before I embarked on this PhD journey. Thank you for making it happen, for believing in my potential, despite the goofy first impression I left. Thanks for making New York, San Francisco, CERN, NASA, ... all of it, a reality, and thanks all the great people I met through you. I will not forget our "tell it to your Teddy Bear" sessions, those reality checks every time I got stuck. Nevermind the awkward name, the technique is ingenious. And thanks to the smart geeks from the Grid group I had the honor to be part of: David, Ronald, Sven, Paco, Mischa, Oscar, Dennis, Jan Just, Tristan, Hurng. I learned a lot from you. Keep on rocking!

Henri, my gratitude goes to you, for the freedom you gave me to go for all the (un)successful ideas I had. Thank you for all your questions, your trust, criticism and patience with my chaotic thoughts. For accepting my failures, and praising my success. Kees, your reviews were invaluable, and so was your humor, and your midnight DAS4 attempts to tame the model checking beast. Thank you for that.

Tim, I guess it's somehow your destiny to end up working with & around awkward Macedonians. We can blame Dragan Bosnacki for that, this time. There are so many emails with silly braindumps I wish I could "undo send", yet you did not give up on me. Even during your sabbatical, with that poor hotel WiFi and your chopped robot voice. Thank you for your "I'm always up for a challenge" 5 years ago, the rest is history. I am indebted to Sonja Georgievska. If it wasn't for you, none of this would have had the slightest chance of happening. Thank you for making Holland a reality! Thank you for your sarcasm, really. Thank you for accepting me homeless with a lost 30kg luggage, on day one in Eindhoven. And introducing me to Zarko, Jasen, Natasa, Meri. A heartwarming hospitality bunch, if only they would migrate to the north also :P

Marcel, I have known you from the first day I set foot at Nikhef. Your laughter is genuine, I can hear it from miles. Thank you for all your efforts in making my first steps at CERN smooth. I met some fantastic people there. Philippe, Joel, Marko, Stefan, Fede, Vladimir, Ricardo, Adri: I never expected to feel at home so quickly, you took me like one of your own, you dear, warm, prodigious people, with a remarkable sense of humor! LHCb is the coolest of all LHC experiments, and I'm not just saying. I mean it. You are the bleeding edge. I learned so much from you. Rarely do I meet people who are so knowledgeable and modest at the same time. I miss you very much. Chris, my dear friend, thank you for making the Geneva stays enjoyable! Underground, the pit, hallways, highways, movies, brainstormings, New York, wine-tasting, guitar plays, Vijay on the radio (that we somehow missed). So much history, I hope our paths cross again.

Niksa....my dear Niksa. You know me too well. Thank you for all the "therapy sessions", all those wine & philosophy late night talks. You gave me a short escape, each time I needed one. You have your original way around my stubbornness,...hey, you even managed to get me off cigarettes. You are such a beautiful mind. Ana, Gordana, Ratko, Tanja, Sonja, Saska, Kiril, Igor: I have known you for what seems like ages. You have shaped me up as a person, and (perhaps too eagerly) put me on a pedestal. You make the trips to Skopje worthy. We may not see eachother often, but when we meet, it is nothing but love.

Ognen, I look up to you, you are my hero. The positive energy you spread is contagious. If I start listing all the things I'm thankful for, I will need an extra chapter. Jose, Bojana, Svetle, Ivana... my Macedonian gang, thank you for making me feel at home in Amsterdam. I'm so proud of all of you, and I feel lucky to have you in my life. No, I'm not being cheesy, you are spectacular. Riste and Irina: I really cherish the great times with you here, and I'm indebted to both, for your moving help, great Macedonian dinners, and witty advice in desperate times. Amsterdam feels emptier now (damn you, Germany!), but I am still secretly looking for ways to bring you back. Nikola and Bojan, my Ijburgers, we shall rediscover the Amsterdam squats together, I promise you. Sorry for the quarantined weekends in the past months, and thank you for those retro-gaming nostalgic moments.

Ana, your love at first sight is a mutual one, just so you know! Funny how smoking breaks can be the causality of many things, such as starting an incredible friendship. You were always late, but worth the wait. Philip, you social butterfly, thanks for being my Friday Night Skate support (literally), a party initiator so many times, and for the unforgettable boat tours around Amsterdam. Albana, Dirk, Bebe, David, Ben, Corina, Otto,... thank you for all the sleepless nights, for all the hilarious Cards Against Humanity games, barbecues, and all bottles of Prossecco reserved for my consumption. I seem unable to sustain my smile when I'm around you. No matter how bad the day has been, you can turn it into a blast. My Amsterdam

family, thank you for all intellectually inspiring conversations and advice. This journey would have been a boring one without you. Ry, *sonce moe*, thank you for your moral boost, your endless support and occasional distractions. For all the warm meals you cooked for me, and all Netflix Friday nights. Thank you for for putting up with my thesis frustrations, and believing in me.

Мама, тато, Фроси: ви благодарам на безрезервната поддршка и трпението со овој мој докторат. Ви благодарам на сите совети. Горда сум и се огледувам на вас, во секој поглед.

Finally, I would like to thank all the members of my thesis committee, for their valuable thesis reviews, dedication, and patience with the scheduling of my defense day.

Daniela
Amsterdam, 2016

Contents

List of Figures

List of Tables

List of Abbreviations

CERN	European Organization for Nuclear Research
LHC	Large Hadron Collider
WLCG	Worldwide LHC Computing Grid
LHCb	Large Hadron Collider beauty experiment
DIRAC	Distributed Infrastructure with Remote Agent Control
WMS	Workload Management System
SMS	Storage Management System
VO	Virtual Organization
LTS	Labeled Transition System
ACP	Algebra of Communicating Processes
PBES	Parametrized Boolean Equation Systems
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
UML	Unified Modeling Language
CASE	Computer Aided Software Engineering
SM	State Machine
AD	Activity Diagram
SD	Sequence Diagram
OCL	Object Constraint Language
PSC	Property Sequence Charts
LSC	Live Sequence Charts
OMG	Object Management Group
PSP	Property Specification Patterns
SOS	Structural Operational Semantics

Chapter 1

Introduction

Software systems are becoming increasingly complex, growing both in code base size and number of involved components. In effect, it becomes more challenging to detect certain design errors before it is too late, and these manifest in financial or human life losses. Testing [69], or running the software under different input stimuli in order to examine different possible behaviors and outputs, is a common form of verification in the software engineering life-cycle. One essential shortcoming of testing is that it is not exhaustive, i.e., cannot guarantee that all relevant behaviors have been covered with the tests. In the famous words of Dijkstra [58], *program testing can be used to show the presence of bugs, but never to show their absence*. This limitation is exacerbated in concurrent systems, where the order of events from various components is unpredictable and difficult to test or reproduce in a controlled manner. Certain design errors in such complex systems (e.g., deadlocks, race-conditions) are notoriously difficult to find purely by means of testing, simply because the number of different possible scenarios is typically prohibitively large.

Formal methods [48] offer more rigorous means of system verification. They should be considered a supplemental, rather than a replacement technique to testing and other means of verification, which contribute to improved quality of the system design and implementation. These mathematically-rooted techniques to prove system correctness, or the absence of system errors, have been a topic of research for several decades. A range of different approaches and supporting software tools have emerged over time, including theorem proving [144; 143], static analysis [173; 18; 54], and model checking [47], to name but a few. Despite the ongoing research effort, they are still considered somewhat exotic and are often met with resistance in many software industry circles. One of the major barriers to the wide acceptance of formal methods is the learning curve: engineers and programmers typically lack the training to use them, while managers are not aware of the context in which such methods should be applied, nor the potential benefits

of using them. An additional obstacle is the fact that their use typically amounts to maintaining two separate model implementations: a formal one, amenable to verification, and a "real" production one. On the other hand, proponents of formal methods claim that their application can substantially improve the software quality and reduce the costs and the risks of failure in the long run [36; 198; 86]. The reality is most likely somewhere in the middle. What started out as manual labor-intensive proof construction using axioms and inference rules, geared towards small sequential programs, has certainly moved far beyond merely adding program assertions to perform runtime program checks. In contrast to three decades ago, academia today is no longer just advocating verification with theoretical results; industry actually applies it, though still rather selectively. The root cause behind this limited penetration of formal methods in the broader industry has itself been a topic of research and debates [31; 146; 28].

One highly effective verification technique is model checking [47; 50; 156]. The rising success of this field stems from the availability of actively-maintained mature tools which automate the algorithmic procedure of determining whether an abstract mathematical model representation of the system satisfies certain behavioral properties, or requirements. These model checking tools expect a model expressed in some formal notation (such as a process algebra [17] or Petri Nets [160]), and a property to be verified, typically expressed as a formula in temporal or modal logic [49; 85]. These logics contain temporal operators such as "next", "eventually", "always", and "until", which allow for reasoning about behavior over time. Simply put, the entire state space of the model is automatically generated while all possible scenarios are exhaustively explored, making the technique equivalent to exhaustive model testing. The state space is typically represented by a state-transition graph (specifically, a Kripke Structure or a Labeled Transition System), a structure containing all reachable states (system snapshots containing values of all variables, including control stacks) and transitions (actions that change the system state). As a result, if an affirmative answer is given concerning a model and a property specification, one can be confident that the model does not contain problematic behavior with respect to the specified property. A strong point of model checking tools is their capability to produce counter-examples: traces leading to the situation that violates the checked property. This way one can get a detailed understanding of the root cause of the detected problem.

There are success stories [106; 111; 153; 193; 194; 18] of applying model checking in industrial research labs, and plenty of communication protocols and safety-critical modules have been put under scrutiny [104; 145; 77; 107; 16; 113; 30], unveiling bugs undetected before. Tools for automating such analysis are well supported and continuously improving (e.g., SPIN [108], NuSMV [44], mCRL2 [88], CADP [81], LOTOS [187]), but there is still a need to close the gap by involving the end-users

as stakeholders, and not only the research community and the theoretical computer science experts in the process. Further effort is required to reduce the barrier to practical use of most such tools, as well as to integrate them into the typical development environments and frameworks used in practical software engineering. This dissertation describes research that aims to achieve that.

1.1 Context and Research Objectives

Having emphasized the need for slotting formal methods into common industrial practice, we formulate our research goals through several research questions. First and foremost, we want to check the feasibility of using formal methods in a specific context. We are motivated by the distributed grid framework called DIRAC [186] (Distributed Infrastructure with Remote Agent Control), developed and used by the LHCb [1] community at the European Organization for Nuclear Research (CERN). It belongs to a class of large-scale distributed systems, whose behavior is primarily driven by data. We therefore focus on distributed object-oriented systems whose behavior is highly data-dependent. Object-oriented engineering [78] is based on a popular paradigm where a software system is composed as a collection of individual entities (called objects), each with a state and behavior, collaborating with one another to achieve the system behavior. Distributed systems are by definition concurrent, and as such are great candidates to benefit from model checking. Unlike hardware, communication protocols, or device drivers, distributed object-oriented systems are typically less structured, asynchronous, have a data-driven behavior, and this restricts and influences the choice of a formalism for modeling them. With that in mind, we aim to address the following research questions.

Research Question 1: *Is there a formalism suitable and rich enough for modeling and addressing the design errors of realistic-scale distributed data-driven systems?*

The existence of such a formalism is necessary, but not sufficient for adopting a formal verification technique such as model checking. The benefits over using more traditional and common quality assurance methods must be explicit and clear.

Research Question 2: *What are the true advantages of using model checking in addition to testing, as part of the widespread software engineering life cycle, in the context of realistic-scale distributed data-driven systems?*

Constructing a formal model and formalizing the properties to be verified is a time-consuming task, and can often be done only by formal methods experts who are fluent in formal languages and temporal logics. Choosing the right level of ab-

straction is far from trivial: keeping too many details about the real system will lead to a state-space explosion, preventing the final verdict from the tool due to limited resources, while leaving out essential behavior may potentially hide problems in the real system or lead to false-negatives being reported. Model checking results are not trustworthy if the model does not reflect the system behavior "accurately enough." Furthermore, interpreting the model checking traces back to the original system behavior often requires the ingenuity of the experts who built the model. While the extra costs (in time and manpower) of using formal methods may be acceptable in research environments or safety-critical systems, this is not always the case in broader contexts and domains. Thus, before model checking can become a widely adopted push-button technology in the broader computing industry, these aspects must be automated as much as possible. This brings us to

Research Question 3: *Is it possible to integrate model checking into the common software development cycle of realistic-scale distributed data-driven systems, by automating the aspects that require formal methods expertise, mentioned above?*

We further refine the last question into more specific ones, to provide a framework for the research objectives and thesis contributions:

- **Research Question 3.1**: *Is it possible to automatically derive target formal behavioral models, and if so, what is the appropriate source model abstraction level?*
- **Research Question 3.2**: *How can we derive formal descriptions of the desired system properties, based on informal software requirements descriptions?*
- **Research Question 3.3**: *Can formal property descriptions and verification results be presented in a way that is understandable and easy to interpret for non-experts in formal methods?*

Software implementations in high-level programming languages provide a detailed view of the system behavior, but already contain too many implementation language-specific details to serve as a footprint for deriving formal models which can be efficiently checked. In addition, verifying implementations directly has mostly shown impractical or impossible (with a few exceptions of verifying ANSI-C [45; 109; 26] and restricted Java code [194; 53]). The limitation of such approaches is that they are necessarily confined to one particular implementation language. Thus, as already indicated, abstractions play a crucial role, but at the same time handcrafting the transformations to more abstract models is risky and error prone. Instead, we can utilize higher-level software models, frequently used as part of the software design process and hence possibly already constructed. Such models are typically

described and documented using visual modeling languages, and play a key role in the model-based design and development (MBD) approach, which aims to support the communication and development cycle in designing systems. Currently the most widespread general-purpose modeling language in software and systems engineering is UML [90]. Over the past decade it has become a "de facto" standard for modeling the static and dynamic aspects of software designs. Class diagrams show the collection of classes that serve as footprints for the object-oriented implementation, while behavioral diagrams such as state machines and sequence diagrams are used to model the dynamics of a system. The overall behavior of distributed object oriented systems can be naturally seen as concurrent sequences of synchronous or asynchronous interactions between components (objects), and sequence diagrams capture exactly this aspect. The abstract syntax of UML is described in terms of the UML metamodel [91], but the semantics [92] is largely defined in plain English. This inevitably causes ambiguities in the interpretation of models, which also prevents their automated analysis and verification.

1.2 Main Contributions

We summarize the contributions of this dissertation in the following statements:

- Our feasibility study of systematically abstracting and modeling subsystems of the DIRAC framework indicates that the mCRL2 formalism [87] is suitable to construct and verify such systems, since it supports the necessary concurrency, data description and data manipulation mechanisms. The mCRL2 process algebra is accompanied by a toolset [88] for model simulation, state-space generation, reduction, visualization, and verification of behavioral properties. Using the toolset, we have discovered critical race conditions and livelocks which were confirmed to occur in DIRAC's implemented logic in production.

- We take sequence diagrams as a starting point for deriving formal models in mCRL2. By attaching a formal, mathematical semantics to UML interactions, we can provide their unambiguous interpretation, and use the advantages of tool-supported automated verification. We introduce such a mapping to facilitate the verification of UML designs with the mCRL2 toolset. The translation preserves the object-oriented aspects of the original model, enabling a round-trip approach in which the verification counter-examples can also be visually presented as sequence diagrams. Through case studies of applying this methodology to DIRAC components, we provide some empirical evidence on the correctness and usefulness of the translation.

- Model checking application specific properties in mCRL2 requires the use of

the μ-calculus [85] modal logic to formally express them. This logic is very expressive, but having to manually write formulas puts an additional burden on the software engineer to master yet another language. To bring the process of correctly eliciting behavioral properties closer to software engineers, we introduce a property assistant tool, as part of a UML-based front-end to the mCRL2 model checking toolset. Based on a well-known property pattern classification [67], which we extend with new patterns, the tool provides assistance to non-experts in eliciting properties which capture the required behavior in real-world settings.

- As a further external evaluation, we surveyed works that use μ-calculus to express system properties from different domains. Our purpose was to reassess the usefulness of the pattern-based classification in event-based systems settings, with a focus on the μ-calculus as a target formalism, as well as to get an indication of how error-prone the manual elicitation of properties can be, even when it has been thoroughly reviewed by formal methods researchers. This survey strengthens the claim of Dwyer et al. [67] that very few patterns are sufficient to express the majority of properties. One implication for users is that it suffices to learn how to apply a small fragment of the pattern templates, to capture the majority of properties. We conjecture that this in turn facilitates faster and easier learning of the corresponding small fragment of the target specification formalism.

1.3 Dissertation outline

The rest of this dissertation is organized as follows. In **Chapter 2** we provide some background information on DIRAC, the software framework for distributed computing that we use in our case studies throughout the remaining chapters. **Chapter 3** tackles **Research Question 1**. We discuss the challenges of modeling large-scale distributed systems such as DIRAC, in order to address concurrency issues arising from the use of shared storage and inter-process communication. We establish some general guidelines on how to obtain a formal mCRL2 model from a system implementation that is scarcely documented, formulate application-specific properties, interpret the model checking results, as well as apply some pragmatic means to tackle state space explosion. In **Chapter 4** we first revisit the syntax and semantic variation points of UML sequence diagrams, and explain how the approach differs from the existing related body of work. Addressing **Research Question 3.1**, and partly **Research Question 3.3**, we present a methodology for an automated translation of UML designs comprising sequence diagrams into the mCRL2 process algebra, which facilitates automated verification and intuitive graphical view and

understanding of the verification results. We demonstrate the applicability of the approach with a case study on DIRAC's core workload system functionality. To make the process of correctly eliciting and formalizing behavioral properties with μ-calculus accessible to non-experts, thereby addressing **Research Question 3.2**, in **Chapter 5** we introduce a property assistant tool PASS. Apart from automated formula synthesis, in the elicitation process, the tool presents the user with a natural language summary and a sequence diagram depicting the behavioral property, that way partly addressing **Research Question 3.3**. In addition, for a certain class of properties, it generates a monitor automaton for runtime verification, or lightweight bug-hunting. A larger case study is provided to evaluate the usefulness of the tool. We further survey existing research on μ-calculus properties specification, and provide some evidence on the usefulness of the property pattern classification, as well as our extensions, in this context. **Research Question 2** is addressed throughout the case studies of Chapter 3, 4, and 5. Finally, we conclude in **Chapter 6**.

A significant part of the content of this dissertation is obtained from the following scientific publications:

1. [165] Daniela Remenska, Tim A.C. Willemse, Kees Verstoep, Jeff Templon, and Henri Bal. Using model checking to analyze the system behavior of the LHC production grid. *Future Generation Computer Systems*, 29(8):2239–2251, 2013. doi: 10.1016/j.future.2013.06.004. (Chapter 3)

2. [166] Daniela Remenska, Jeff Templon, Tim A.C. Willemse, Philip Homburg, Kees Verstoep, Adria Casajus, and Henri Bal. From UML to Process Algebra and Back: An Automated Approach to Model-Checking Software Design Artifacts of Concurrent Systems. In *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 244–260. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-38088-4_17. (Chapter 4)

3. [167] Daniela Remenska, Tim A.C. Willemse, Jeff Templon, Kees Verstoep, and Henri Bal. Property Specification Made Easy: Harnessing the Power of Model Checking in UML Designs. In *Formal Techniques for Distributed Objects, Components, and Systems*, volume 8461 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-662-43613-4_2. (Chapter 5)

4. (submitted: Dec., 2014; under review) Daniela Remenska, Tim A.C. Willemse, Kees Verstoep, Jeff Templon, and Henri Bal. Assisting Non-Experts in Property Specification for Automated Formal Verification. In *IEEE Transactions on Software Engineering*. (Chapter 5)

Chapter 2

DIRAC: A Community Grid Solution

2.1 Introduction

The focus of this thesis is on integrating model checking into the common software engineering cycle of realistic-scale distributed systems. The work was primarily motivated by a real-world software called DIRAC (**D**istributed **I**nfrastructure with **R**emote **A**gent **C**ontrol) [186; 185], developed and used by the LHCb [1] community at the European Organization for Nuclear Research (CERN). DIRAC is relatively well-engineered software, and its distributed and large-scale nature provides good chances for any potential functional problems and inconsistencies to materialize in real life. The case studies presented throughout the thesis are based on actual DIRAC components, and in the following we introduce the main concepts necessary to understand them. We restrict the thesis focus to addressing functional aspects in distributed systems; in particular behaviors which can result in deadlocks, race conditions, and other concurrency issues, driven by some of the ones indicated in this chapter. DIRAC is not a real-time system with hard deadline constraints for event completions, although its efficiency and scalability are of a great concern, and are constantly improving. We therefore abstract from timing, probabilities and other quantitative aspects which could be analyzed in another setting.

The Large Hadron Collider beauty (LHCb) experiment is one of the four large experiments conducted on the Large Hadron Collider (LHC) accelerator. Immense amounts of data are produced at the LHC accelerator, and subsequently processed by physics groups and individuals worldwide. The sheer size of the experiment is the motivation behind the adoption of the grid computing paradigm, to process the data efficiently. The grid storage and computing resources for the LHCb experiment are distributed across many national and local scientific institutes around the globe. To cope with the complexity of processing the vast amount of data, a generic grid

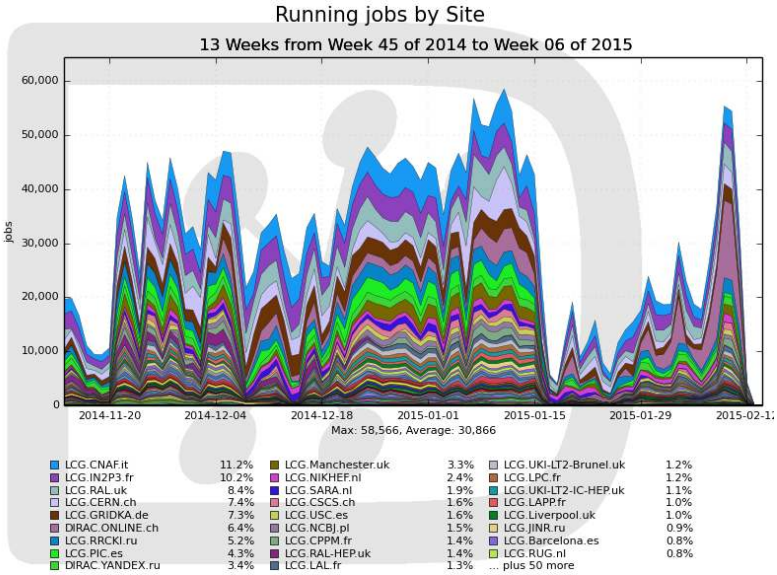


Figure 2.1: LHCb job execution rate by site [1]

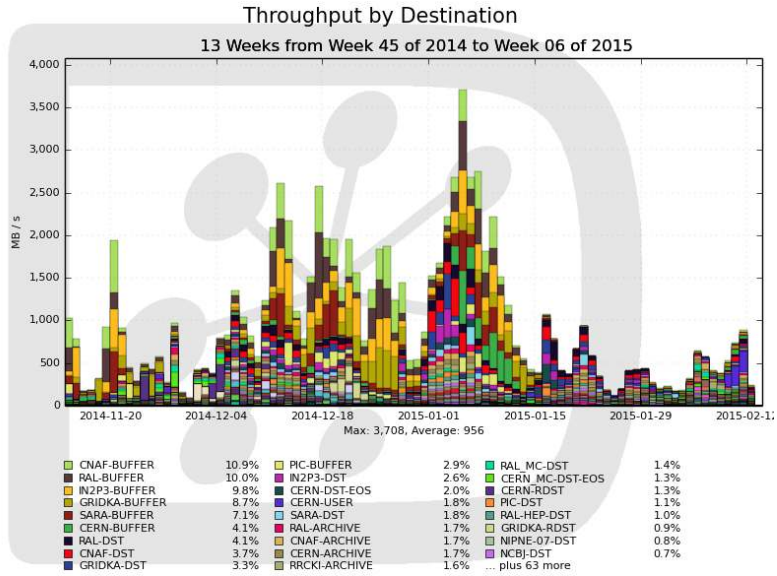


Figure 2.2: LHCb data transfers throughput [2]

system - DIRAC, was designed and developed initially for the needs of the LHCb community. It is a software framework for distributed computing, that forms a layer between user communities and the heterogeneous distributed computing and

[1] Source: LHCbDIRAC web portal https://lhcb-portal-dirac.cern.ch/DIRAC/
[2] See footnote 1

storage resources, to allow for an integrated, optimal, and reliable usage of these resources.

The development of DIRAC started in 2002 as a system for producing simulation data that would serve to verify physics theory, aspects of the LHCb detector design, as well as to optimize physics algorithms. It gradually evolved into an extensive generic grid system for data and job management, based on a general-purpose framework that can be reused by communities besides LHCb. Today, it covers all major LHCb tasks starting with the raw data transfer from the experiment's detector to the grid storage, several steps of data processing, up to the final user analysis. It uses a CPU power of over 200 kHS06[3] concurrently, and 20M file replicas distributed over hundreds of grid sites, provided by the WLCG computing grid [27]. The community of users has grown to several hundreds, loading the LHC grid resources with over 50K simultaneously running jobs (Figure 2.1) and over 100K jobs executed per day, during peak processing periods. More than 25PB of data is stored and visible in the LHCb file catalog, with daily data transfers between storage units reaching 3.5GB/s (Figure 2.2).

Python was chosen as the implementation language, since it enables rapid prototyping and development of new features. DIRAC follows the Service Oriented Architecture (SOA) paradigm, accompanied by a network of lightweight distributed agents which animate the system. Its main components are depicted in Figure 2.3.

The *services* are passive components that react to requests from their clients, possibly soliciting other services in order to fulfill the requests. They run as permanent processes deployed on a number of high-availability hosts, and store the dynamic

[3]HEP-SPEC06 benchmark: http://w3.hepix.org/benchmarks

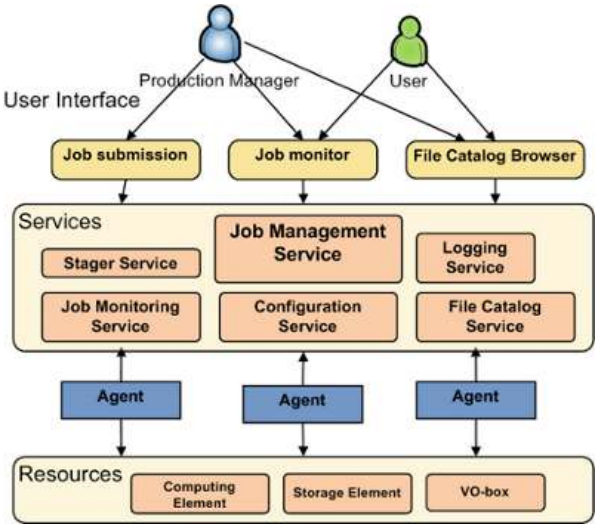


Figure 2.3: DIRAC Architecture overview

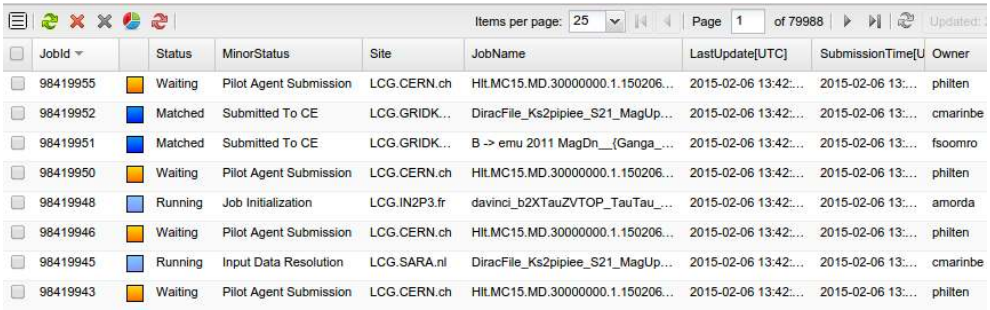


Figure 2.4: LHCbDIRAC web portal

system state information in database repositories. The user interfaces, agents or running jobs can act as clients placing the requests to DIRAC's services.

Agents are active components that fulfill a limited number of specific system functions. They run in different environments, depending on their mission. Some are deployed close to the corresponding services, while others run on the grid worker nodes. Examples of the latter are the so-called Pilot Agents, part of the Workload Management System explained in the following section. All DIRAC agents repeat the same logic in each iteration cycle: they monitor the service states, and respond by initiating actions (like job submission or data transfer) which update the states of various system entities. The logic of each individual agent is kept relatively simple; the overall system complexity emerges from the cooperation among them. Namely, these agents run concurrently, and communicate using the services' databases as a shared memory (blackboard paradigm [140]) for reading and updating the states of various entities.

Resources are software abstractions of the underlying heterogeneous grid computing and storage entities allocated to LHCb, providing a uniform interface for access. The physical resources (disks, data archival tapes, CPUs) are controlled by the grid site managers and made available through middleware services such as gLite [126]. The DIRAC functionality is exposed to users and developers through a rich set of command-line tools forming the DIRAC API, complemented by a Web portal (Figure 2.4) for visually monitoring the system behavior and controlling the ongoing tasks. Both the Web and command-line *interfaces* are based on secure system access using X509 certificates.

At the time of writing, the most powerful DIRAC hardware configuration is the one used by LHCb: it consists of about 20 servers located at CERN, hosting the central DIRAC services, including 6 MySQL database servers. A small fraction of the servers and databases are used for testing and certification purposes, and are connected to the same pool of WLCG computing and storage resources. Several other experiments are currently using DIRAC on an every day basis, with new user groups (virtual organizations, or VOs) evaluating its functionality and performance

before adopting it in production. Setups are also shared between smaller VOs which do not have the manpower to maintain them on their own.

Although much effort is invested in making DIRAC reliable, entities occasionally get into inconsistent states, leading to a loss of efficiency in both resource usage and manpower. Debugging and fixing the root of such encountered behaviors becomes a formidable mission due to multiple factors: the inherent parallelism present among components deployed on different physical machines, the size of the implementation (∼150,000 lines of Python code), the distributed knowledge of different subsystems within the collaboration, and the limited manpower available. Depending on the frequency of certain heavier data (re)processing campaigns throughout the year, the occurrence of such inconsistencies can be up to several reported cases per month. They can have a blocking effect on these grid activities, some of which are with a strict deadline. However, locating and fixing the root cause of such a problem sometimes has a lower priority within the collaboration, especially if it requires substantial time for a deeper analysis. Typically, if possible, a temporary alleviation is done instead, like manipulating database records or restarting processes manually, in order to quickly "unblock" production activities.

In the following, we focus on two related subsystems that are considered the backbone of DIRAC. These are the ones where problematic state changes, which are difficult to trace and correct, are frequently encountered. Understanding their behavior is essential for interpreting the issues that were discovered during the model analysis and verification.

2.2 Workload Management System

The main component of DIRAC is the Workload Management System (WMS), responsible for managing the workflow of jobs submitted to the grid. Taking into account the heterogeneous, dynamic, and high-latency nature of the distributed computing environment, a *Pilot Job paradigm*, illustrated in Figure 2.5, was chosen as an efficient way to implement a pull scheduling mechanism. Pilot jobs are simply resource reservation processes without an actual payload defined a priori. They are submitted to the grid worker nodes with the aim of checking the sanity of the operational environment just before pulling and executing the real physics processing payload. This increases the job success rate, from the perspective of end users. Centralized *Task Queues* contain all the pending jobs, organized in groups according to their requirements. This enables efficient application of job priorities, enforcing fair-share policies and quotas, which are decided within the LHCb community, without the need of any effort from the grid sites.

Originally, the DIRAC WMS submitted pilot jobs using the native gLite middleware tools, but this method was shown to be suboptimal [147]. The gLite resource

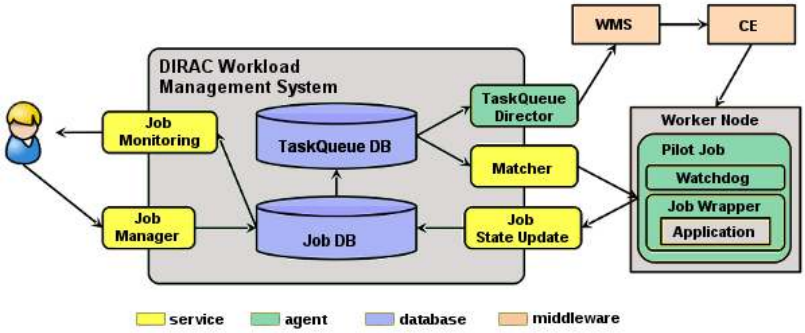


Figure 2.5: DIRAC Workload Management System [158]

brokers are centralized components, and as such they do not have the capacity and flexibility to cope with the load of a community such as LHCb. Using multiple resource brokers, rather than a single one, was not the solution: all brokers use the same site state information and would always submit jobs to the most suitable one globally, causing overloads to certain sites while underloading others very often. In addition, the gLite information system was introducing a significant time lag in propagating changes in site status information to the brokers. With the wide deployment of the CREAM (Computing Resource Execution And Management) [9] service for job management operation at the Computing Element (CE) level, DIRAC's WMS switched to submitting pilot jobs directly to sites. By interrogating the status directly, the information propagation latency was greatly reduced, making the job distribution across sites more balanced and the job turnaround faster.

The basic flowchart describing the evolution of a job's states is depicted in Figure 2.6. After submission through the *Job Manager* ("Received"), the complete job description is placed in the DIRAC job repository (the Job DB, see Figure 2.5). A chain of optimizer agents checks whether the jobs are eligible for execution ("Checking"), and prioritizes them in queues, utilizing the parameters information from the Job DB. Although the grid storage resources are limited, it is essential to keep all data collected throughout the experiment's run. Tape backends provide a reliable and cheap alternative for permanent data archival. In case a job needs to process files already archived on tape, there is an additional workflow step in which the *Job Scheduling Agent* (of the optimizers chain) passes the control to a specialized *Stager* service (part of the Storage Management System elaborated in the following section), and jobs acquire a "Staging" status. Once the optimization is completed, the job is placed in a task queue ("Waiting"). Based on the complete list of pending payloads, a specialized *Task Queue Director* agent submits pilots to the computing resources via the available job submission middleware (historically the gLite centralized WMS service, nowadays the CREAM CE one at each site). After a *Matcher* service pulls the most suitable payload for a pilot ("Matched"), a *Job Wrapper* ob-

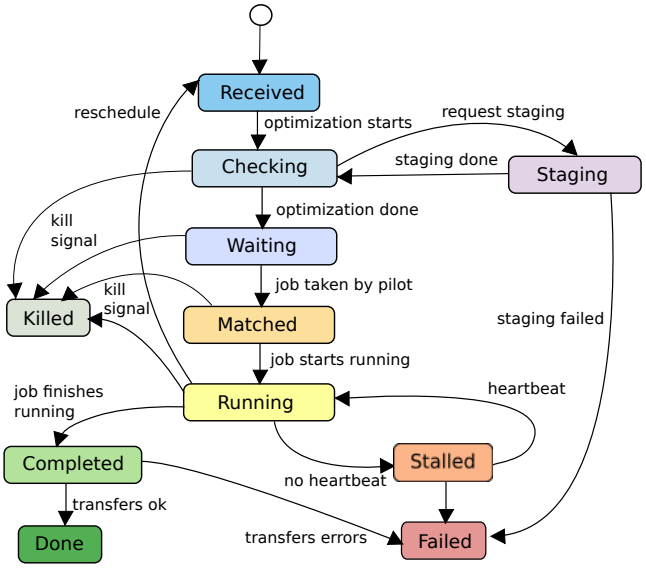


Figure 2.6: DIRAC Job state machine

ject is created on the worker node, responsible for retrieving the input sandbox, performing software availability checks, and executing the actual payload on the worker node ("Running"). Finally, a "Completed" job may have output data which is necessary to upload to a permanent storage, and this step can succeed ("Done") or fail ("Failed") for various reasons. The wrapper can catch any failure exit state reported by the running physics applications. At the same time, a *Watchdog* process is instantiated to monitor the behavior of the Job Wrapper and send heartbeat signals to the monitoring service. It can also take actions in case resources are soon to be exhausted, the payload stalls, or a management command for killing the payload is received. It should be noted that jobs also have a so-called *Minor Status*, which is a more descriptive addition to the main job status, but also necessary for the functionality of the optimizers part of the workflow.

The WMS has been a subject of evolution and improvement, as we will see in Chapter 4. In particular, the chain of optimizer agents revealed certain weaknesses during periods of intensive production activities. Their polling mechanism for discovering new jobs to handle from the central database was not reactive enough. Over the past few years, a more scalable event-driven solution has been in development [157], and later eventually put in production. New DIRAC components, called *Executors*, were designed to handle the incoming jobs in an event-driven manner, organized around a *Dispatcher* service responsible for pushing job processing work to these Executors. This has shown to improve the reactiveness of the WMS when under heavy load.

2.3 Storage Management System

The DIRAC SMS provides the logic for pre-staging files from tape to a disk cache frontend, before a job is able to process them. Smooth functioning of this system is essential for production activities which involve reprocessing older data with improved physics software and happen typically several times per year.

A simplified view of the system is shown in Figure 2.7. The workflow is initiated when the *Job Scheduling Agent* detects that a job is assigned to process files only available on tape storage. It sends a request for staging (i.e., creating a disk replica) to the *Storage Manager Handler* service with the list of files and a callback method to be invoked when the request has been processed. This information is stored by the service in the *StorageManagement DB*, and is subsequently processed by a number of agents in an organized fashion. The Entity-Relationship model of this database is shown in Figure 2.8. The relevant entities are *Tasks*, *CacheReplicas*, and *StageRequests*, maintaining states observed and updated by the agents. *Tasks* contain general information about every job requesting a service from the SMS, while *CacheReplicas* records keep the details about every file (i.e., the storage where it resides, the size, checksum, number of tasks that requested it). There is a many-to-many relationship between these two entities, as multiple jobs can independently request the same file(s) to be staged. Information about each request for staging a file (the actual submission to the site, staging completion, and request expiration timestamps) is stored in a *StageRequests* table.

Figure 2.9 illustrates the state machine[4] of a single instance of the *CacheReplicas* entity. The processing begins with the *Requests Preparation Agent* checking whether

[4]Note that the state machine depictions in this chapter are merely conceptual, and should not be treated formally. They are not always properly reflected in the actual implementation.

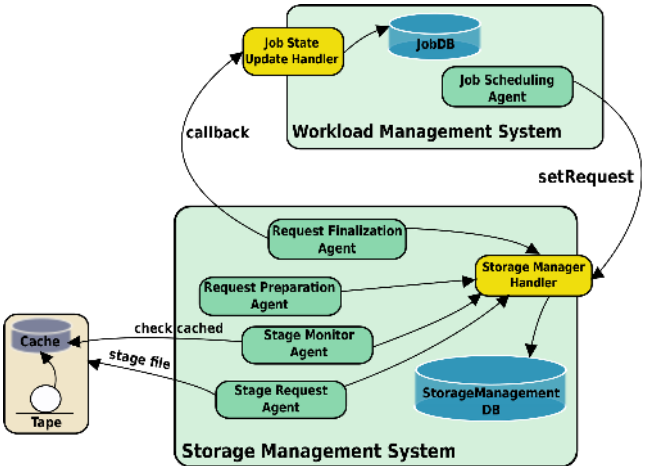


Figure 2.7: DIRAC Storage Management System

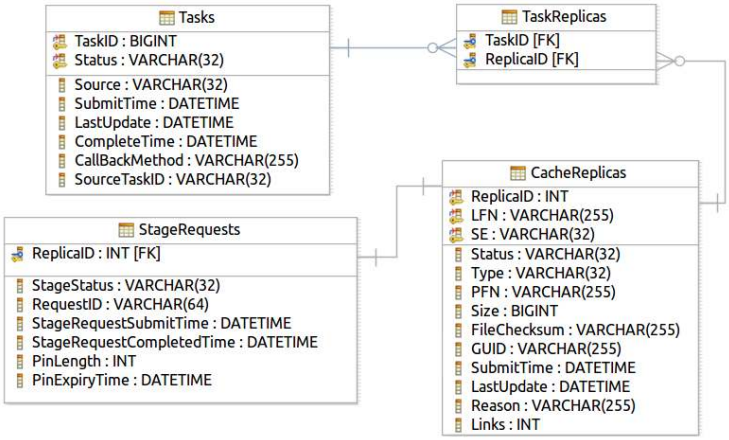


Figure 2.8: Relationships between StorageManagement DB entities

all "New" replica entries are registered in a file catalog which contains file metadata and allows for dataset querying. In case of problematic (or non-existent) catalog entries, it can update their state to "Failed". Non-problematic files are updated to a "Waiting" state. The *Stage Request Agent* is responsible for placing the actual staging requests for all "Waiting" entries, via a dedicated storage middleware that communicates with the tape backends. These requests are grouped by physical storage destination prior to submission, and carry information about the requested (pin) lifetime of the replicas to be cached. Using knowledge about the disk cache frontend capacities at every physical grid location, certain replicas may be temporarily updated to "Offline" (and picked up in some later iteration), if the total storage space needed to stage all requested replicas exceeds those capacities. If certain pathologies are discovered (i.e., lost, unavailable, or zero-sized files on tape), the

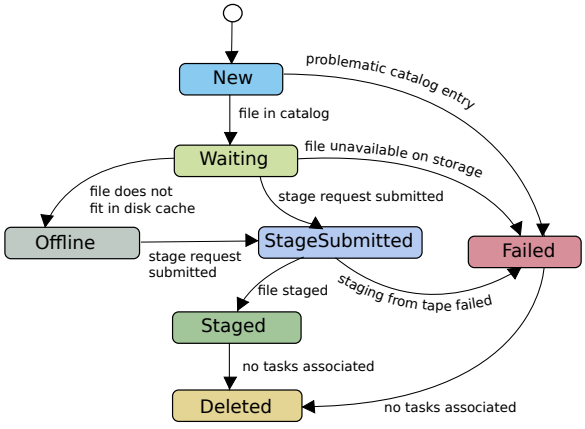


Figure 2.9: CacheReplicas state machine

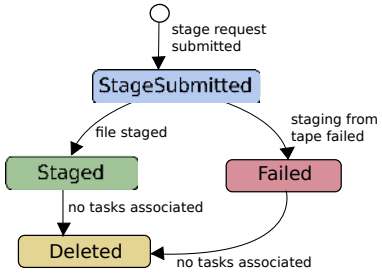


Figure 2.10: StageRequests state machine

agent will update the corresponding entries to "Failed". Otherwise, they are promoted to "StageSubmitted" after placing the actual staging requests. The agent responsible for monitoring the status of submitted requests is the *Stage Monitor Agent*. It achieves this by polling the storage middleware to see if the "StageSubmitted" files are cached on disk. If this is the case, the replica entries are updated to "Staged". Various circumstances of tape or middleware misbehavior can also fail the staging requests. The last one in the chain is the *Request Finalization Agent*. Those tasks for which all associated replicas are "Staged", or for which any replicas are "Failed" (both considered final states), are cleared from the database, and callbacks are performed to the WMS, which effectively wakes up the corresponding jobs, so they can eventually run on the cached replicas. If there are no more tasks associated to particular replicas, the respective *CacheReplicas* and *StageRequests* entries are also removed by this agent. The pseudo-state "Deleted" is not part of the design, but is added here for clarity and consistency with chapters 3 and 5, where we model-check faulty behaviors of unintentional removal and referencing of (removed) tasks and replicas. In practice, all instances of the relevant entities are created and deleted dynamically.

The state machine of a *Tasks* entity instance closely follows the states of the replicas associated to that task. In that sense, the state of the task is always promoted to the "lowest common denominator" state of its associated replicas states, the order being: ["Failed", "New", "Waiting", "Offline","StageSubmitted", "Staged"]. For instance, if all but one replicas are "StageSubmitted", and the remaining one is still "Waiting", the status of the corresponding task will be "Waiting". However, if any of its replicas are declared "Failed", then the task is immediately declared "Failed" as well. *StageRequests* instances have a simpler state machine, containing a subset of the states and transitions of the *CacheReplicas* one. They are intentionally kept as separate entities. The reasons for this are implementation-specific, and will become more clear in the next chapter. Once a request for staging a file is submitted, the newly created *StageRequests* entry immediately gets a "StageSubmitted" state. The rest of the behavior also resembles the *CacheReplicas* one, as can be seen in Figure 2.10.

At the time of writing, this subsystem was still in an evolutionary phase. In multiple instances, tasks or replicas would become "stuck", effectively blocking the progress of jobs in the WMS. Tracing back the sequence of events which led to the inconsistent states is non-trivial. To temporarily alleviate such problems, the status of these entries would often be manually reset to the initial "New" state, so that agents can re-process them from scratch. Occasionally, SQL error messages would be reported from unsuccessful attempts of the SMS service to update the state of non-existent table entries.

Chapter 3

Formal Modeling and Analysis of Distributed Systems

3.1 Introduction

In this chapter we deal with the question of how to formally model the behavior of distributed object-oriented systems like DIRAC, so that formal analysis and model checking can be done. For this, we first introduce a well-defined structure known as a Labeled Transition System (LTS), which is a mathematical notation used to model such system behavior. We use the process algebra of mCRL2, a language for specifying LTSs in a concise manner, and the accompanied mCRL2 toolset, for reasoning about them. To investigate the feasibility of using the mCRL2 formalism for capturing the intended system behavior, necessary for model checking certain properties of interest, we use two subsystems as case studies: the Workload Management and the Storage Management System of DIRAC. We devise and apply rules for control-flow and data abstractions, and perform analysis and model checking on the resulting models. As we will see, this already helped to better understand the behavior of these DIRAC subsystems, as well as detect race conditions which were confirmed to occur in production.

3.2 Labeled Transition Systems

Distributed systems such as DIRAC have a behavior commonly represented as an edge-labeled graph called a Labeled Transition System (LTS). The nodes in the graph represent the states of the system, edges between nodes represent an atomic state change and the edge labellings represent atomic events (or actions) such as reading, sending and successful communications within the system. *Behaviors* of a distributed system are modeled by the sequence of edge labels obtained by

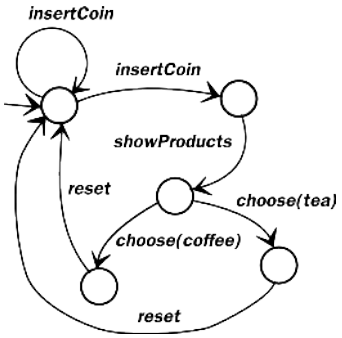


Figure 3.1: Non-deterministic behavior of a vending machine

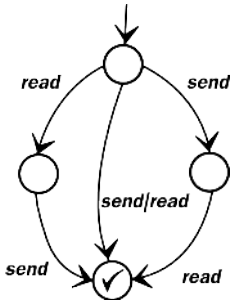


Figure 3.2: Behavior involving multi-actions

traversing along the edges of the graph. Multiple edges emanating from a single node indicate that the state represented by the node may possibly evolve (non-deterministically in case edge labels are identical) in different ways. The state space is the set of states reachable through the transitions in the LTS. Such an LTS example illustrating the behavior of a simple vending machine is shown in Figure 3.1. From the initial state, denoted by an incoming arrow, it is not possible to know whether there is one or multiple *insertCoin* actions before products are offered. [1]

Figure 3.2 illustrates a different scenario. Two distinct actions *send* and *read* can happen sequentially (*send* after *read*, or *read* after *send*) in an LTS, but they can also happen at the same time (concurrently), in which case we write $send|read$ and call it a *multi-action*. In general, multi-actions represent a collection of actions that occur truly at the same time. If an LTS has successfully terminating states, they are indicated with a tick ($\surd$). A state is deadlocked if there is no outgoing transition from it. Below we give a formal definition of a labeled transition system.

Definition 3.2.1. Labeled Transition System **is** a five-tuple $A = (S, Act, \longrightarrow, s_0, T)$ where

- S is a (possibly infinite) set of states
- Act is a set of actions, possibly containing multi-actions
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation
- $s_0 \in S$ is the initial state
- $T \subseteq S$ is the set of successfully terminating states

We commonly use the infix notation $s \xrightarrow{a} s'$ for $(s, a, s') \in \longrightarrow$, where $a \in Act$ and $s, s' \in S$. A *trace* $\rho \in Act^*$ is a sequence $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \cdots \xrightarrow{a_n} s_{n+1}$ for some $n \geq 0$, where $(s_i, a_i, s_{i+1}) \in \longrightarrow$. An empty trace is denoted by ϵ.

In practice, even mundane distributed systems can give rise to rather large LTSs. When modeling systems, typically LTSs are not written explicitly as edge-labeled

[1] This vending machine can in fact take all your money without offering you a beverage. Such is life sometimes.

graphs, but a more concise, high-level mathematical formalism and notation is used, such as a process algebra or a state machine description language. Given such a high-level language for describing an LTS, there is a computation problem of generating this LTS. The number of behaviors represented by the LTS may even be infinite. Even when the generation of the LTS with some software tool is effective, it can be computationally expensive in terms of memory and computation time, as the state space tends to grow exponentially with the number of concurrent process descriptions in the modeled system. This can sometimes be overcome by minimizing the LTS, after it has been generated. Minimization is a (typically automated) procedure for reducing the size of the LTS. Of course, the behaviors relevant to the analysis should be preserved, but then we should be more specific on what are the "relevant behaviors". Depending on the criteria, there are different notions of equivalence between LTSs defined in the literature, leading to different procedures for preserving behaviors under LTS minimization. Many model checking tools have efficient means of detecting whether two systems exhibit equivalent behaviors, and can perform state space reduction with respect to a certain equivalence.

For instance, *strong bisimulation* preserves any potential deadlocks, as well as decision moments (branching) of the original LTS, which other equivalences (e.g. *trace equivalence*) do not. In fact, if two LTSs are equivalent under strong bisimulation, they cannot be distinguished by any realistic form of observation, so they can be considered equal. Consider the two LTSs in Figure 3.3, which are trace equivalent. To give the intuitive argument why these two processes are different, the "The Lady, or the Tiger" story by Frank Stocktonis is often used. An accused subject is put in a public arena with two closed doors. His fate depends on the choice of the door he opens. Behind one of the doors there is a hungry tiger, while the other conceals a princess. To model such a behavior, the opening of the doors, as well as the resulting behaviors, are considered atomic actions. The correct model is shown on the left side of the figure. It expresses that whether the accused will be confronted with a tiger or with a lady, depends on the *open_door* action. The model on the right is wrong, because after opening a door, the accused can still somehow choose his fate. Under trace equivalence, these two models would be considered indistinguishable (equivalent), as they produce the same traces, namely $open\_door \cdot marry\_princess$

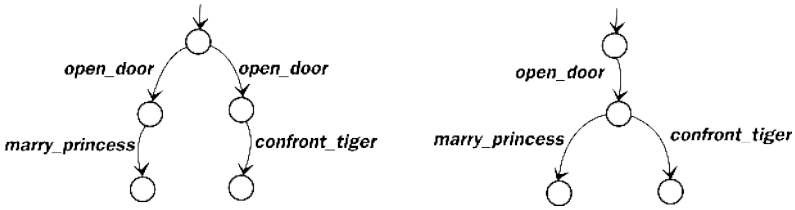


Figure 3.3: The lady, or the tiger?

and $open\_door \cdot confront\_tiger$. However, the behavior of both systems can be experienced to be different, as we will see in Chapter 5.

3.3 The mCRL2 Formalism

The language mCRL2 [87], which extends the algebra of communicating processes (ACP) [24], can be understood as a language for specifying LTSs and reasoning about them. In other words, the semantics associated with the mCRL2 syntax is a Labeled Transition System. In this section we give an overview of mCRL2 that allows for understanding of the modeling approach and the examples we provide in the following sections, but we do not treat mCRL2 (in particular, the formal semantics) in depth. The fragment of the mCRL2 process algebra we are concerned with in this thesis is given by the following BNF grammar:

$$p ::= a(d_1, \ldots, d_n) \mid \tau \mid \delta \mid p + p \mid p.p \mid p||p \mid \sum_{d:D} p \mid c \rightarrow p \diamond p \mid \Gamma_C(p) \mid \nabla_V(p)$$

The basic building blocks of the language are *actions* that can carry data parameters, representing atomic events such as *read* and $send(4)$. The actions terminate successfully. A special constant action *silent step* (τ) is defined, for denoting internal unobservable behavior. *Deadlock* (δ) is a special process which cannot perform any action, in particular it does not terminate successfully. Although not very common, it can be used to explicitly force inaction in a process behavior. *Processes*, representing LTSs, can be composed using operators such as sequential composition and alternative composition to obtain new processes. If processes p and q represent some system, their alternative composition, denoted $p + q$, represents the system that behaves as p when the first action executed comes from p, and it behaves as process q otherwise; this models a *non-deterministic choice* between behaviors. In terms of the underlying LTS model, this may be understood as the operation that

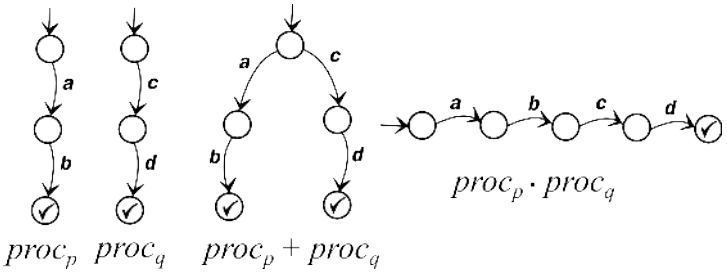


Figure 3.4: Alternative and sequential process composition

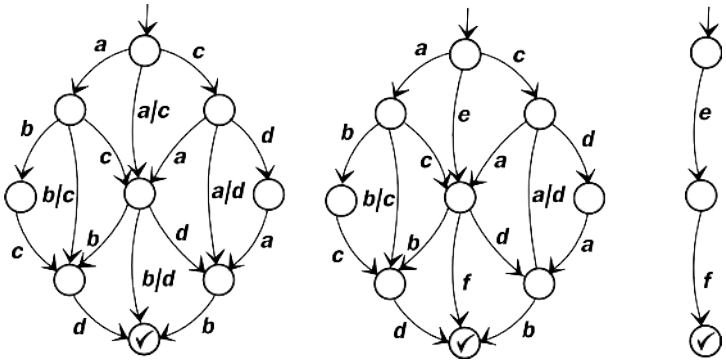


Figure 3.5: Parallel process composition.
Left: $p \mid\mid q$. Middle: $\Gamma_{\{a|c\to e,\ b|d\to f\}}(p \mid\mid q)$. Right: $\nabla_{\{e,f\}}(\Gamma_{\{a|c\to e,\ b|d\to f\}}(p \mid\mid q))$

constructs a new LTS from the LTSs of p and q by creating a new top node that inherits all edges from the top nodes of the LTSs of p and q (Figure 3.4). Sequential composition simply allows to "glue" the behavior of two processes. That is, $p \cdot q$ behaves just as process p, and, upon successful termination, it continues to behave as process q. In the underlying LTS models, this essentially creates a new graph that merges the top node of q's LTS with the nodes marked as terminal in p's LTS. Two processes p and q can be put in parallel using the operator $||$, i.e. $p \mid\mid q$ denotes parallel process composition. The actions in these two parallel processes can happen independently of each other. In other words, an action from process p can happen before, after, or at the same time with an action from q (Figure 3.5 left).

Concurrent processes can interact by enforcing synchronous communication upon certain actions. If actions a and c are intended to happen at the same time (as two ends of a synchronous communication, for instance), rather than as single isolated actions in arbitrary order, this should be specified using the mCRL2 operator *comm* (Γ) in the following manner: $\Gamma_{\{a|c\ \to e\}}$. In practice this means that actions a and c can be observed as a single (multi-)action e, when they happen at the same time (Figure 3.5 middle). All occurrences of $a|c$ will be replaced by e, provided that the data parameters that these two actions may carry, match. The silent step τ can synchronize with any action, i.e., $\tau|a = a$. To explicitly allow those multi-actions from a set V that we want to see as a result of communications within a process p, while implicitly blocking all other actions, the operator $\nabla_V(p)$ is used (Figure 3.5 right).

Actions can carry data values, allowing one to model inter-process communication, exchange of data, and data-based (deterministic) decisions. Examples of actions parametrized with data are $read(1.5)$, $send([1,3])$, and $setAlarm(true, 2)$. Processes can be parametrized by data parameters and values, too. This way, the process behavior can be influenced by the values such parameters carry; mCRL2 of-

fers two operators for this. First, given a process description $p(d)$ in which a variable d can occur, $\sum_{d:D} p(d)$ describes the process which allows for a non-deterministic choice $p(d)$ for any value d from a certain domain D. The $\sum$-operator can best be understood as a generalization of the binary choice ($+$) operator. This operator is mostly used to model the reading of certain data values passed by another process via synchronous communication. Second, the *if-then-else* constructs of the form $b \to p \diamond q$ can be used to describe the process that behaves as process p if the Boolean expression b holds and as process q otherwise.

Apart from describing processes, mCRL2 has a data language part that facilitates describing realistic systems where data influences the system behavior. In the following, we do not give a full exposition of the data language possibilities, only illustrate some of the relevant ones for understanding the rest of the thesis. The data language has built-in support for standard data types such as Booleans (*Bool*), and integer (*Int*), natural (*Nat*), positive (*Pos*), and real (*Real*) numbers, and infinite lists (*List*(_)), sets (*Set*(_)) and bags (*Bag*(_)) over arbitrary data types. In addition, mCRL2 provides various ways in which users can define their own custom data types and operations on these. Using the keyword **sort** one can define one's own enumerated data types and reason about them. For instance, an enumerated type *Semaphore* with elements *red*, *yellow*, and *green* can be defined as follows:

sort *Semaphore* $=$ **struct** *red* | *yellow* | *green*;

Furthermore, auxiliary functions over data types can be declared using the keyword **map**. For example, the *next* and *isBroken* functions can be defined as follows:

map *next: Semaphore* $\to$ *Semaphore*;

map *isBroken : Semaphore* $\times$ *Street* $\to$ *Bool*;

The *isBroken* function takes two arguments (of sorts *Semaphore* and *Street*) and yields an element of Boolean type. To further define how these functions operate, equations can be used with the keywords **eqn** and **var**:

var *sem: Semaphore*;
str: Street;
eqn *next(red)* $=$ *green*;
next(green) $=$ *yellow*;
next(yellow) $=$ *red*;
(next(sem)==sem) $\to$ *isBroken(sem,str)* $=$ *true*;
$\neg$*(next(sem)==sem)* $\to$ *isBroken(sem,str)* $=$ *false;*

The last equation line defines one of the rules on how the *isBroken* function should evaluate, with the following meaning: when the *isBroken* function is applied to data values of the variables *sem* and *str*, it should result in a Boolean value *false* if applying the function *next* on the data value of *sem* results in a value different from the original one of *sem*.

Such features as those of the mCRL2 language outlined above, are indispensable

for modeling distributed and concurrent systems. A more elaborate description of mCRL2 and its features can be found in [57]. Our choice for using the mCRL2 language to model DIRAC's behavior is motivated by its rich set of abstract data types as first-class citizens, as well as its powerful, actively maintained toolset for analyzing, simulating, and visualizing specifications, as we will see.

3.4 From DIRAC to mCRL2

Any formal analysis uses a simplified description of the real system. Even in the best possible scenario, where the target implementation language and the modeling notation are close, it is practically impossible to avoid the use of abstraction to create a simplified model. Software implementations are large and often contain many details that are irrelevant for the intended analysis. In that sense, a model is an abstraction from reality, where certain implementation details are ignored. Abstraction aims at reducing the program's state space in order to overcome the resource limitations [148] encountered during model checking. Furthermore, specification languages describe *what* is being done, abstracting away from the details of *how* things are done. Then, the ultimate question, nicely summarized by David Parnas [146], is: *how do we establish correspondence between model and code?*

High-level structural design documents are a good starting point, but insufficient for building a model that is amenable to analysis of useful behavioral properties. Behavioral properties describe some (un)desired aspects of the system behavior. For instance, it is generally desired that the system is free from deadlocks, or that every locking operation is eventually followed by an unlocking one. In absence of more detailed up-to-date behavioral designs, we based our SMS and WMS formal models on the source code and discussions with developers. A popular abstraction technique for identifying code subsets that potentially affect particular variables of interest (slicing criteria) is program slicing [101], broadly applied to static and dynamic program analysis. It reduces the behavior of a program by removing control statements and data structures deemed irrelevant for the criteria. In the context of building a formal model for the purpose of automatic verification of properties, code slicing can help to focus on parts of the implementation which are relevant for the final analysis, i.e., the intended properties to be checked. If the abstract model is checked and found to be in conformance with the property, one can be sure that the behavior of the original program satisfies this property too. In other words, the obtained abstraction is sound with respect to that property.

To the best of our knowledge, the only practical applications of automated code slicing have so far been limited to a few languages, namely on C/C++ and Java programs ([3; 2; 53]). Research has not yet matured on the topic of code slicing for dynamic languages, such as Python. Therefore, we performed the program slicing

manually, relying on the Eclipse IDE for reverse-engineering and dependency analysis of the subsystems. Given the recurrent invalid state transitions of entities within DIRAC, we considered the possible race conditions caused by multiple agents updating the service states to be the target of our analysis. We limited the scope to the analysis of the following DIRAC entities: *Tasks* (SMS), *CacheReplicas* (SMS) and *Jobs* (WMS). These are the starting points that determine our slicing criteria.

Depending on whether the abstracted underlying LTS is smaller or larger with respect to the original system, there are two general abstraction approaches: over-abstraction and under-abstraction. These can be used in conjunction with code slicing, or when we do not know upfront all the properties we wish to check. Over-abstractions contain more behavior than the original system. This can be achieved, for example, by removing data from certain deterministic branching points in the real system, and introducing non-determinism instead. This reduces the number of states, or the size of the state vector, but increases the number of possible transitions, and hence the number of behaviors. For example, in the simple vending machine in Figure 3.6 (left) which accepts 50 cents and allows one to choose coffee or tea, we can disregard this beverage choice and model both branches as equally possible. Even more, we can sometimes get our beverage without paying enough money for it, as can be seen in Figure 3.6 (middle). Not all properties are preserved by over-abstracting the behavior. For instance, the over-abstracted model can become deadlock-free because of the non-determinism and the extra behavior introduced, even though the original one exhibits deadlock. Nevertheless, such abstractions can be useful for *safety* properties, which state that something bad never happens. If there are no model checking errors of a safety property in the over-abstraction, one can be certain that there is no problematic behavior with respect to that property, in the original system. On the other hand, the discovered errors can be spurious

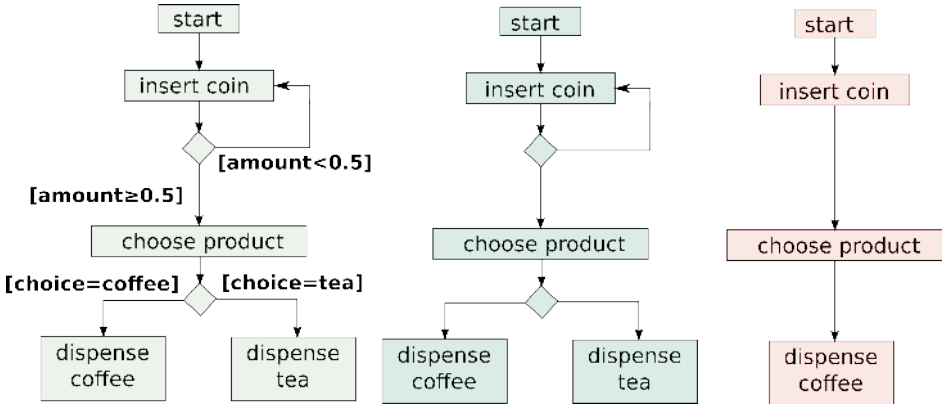


Figure 3.6: Left: Control flow graph of a simple vending machine. Middle: Over-abstraction. Right: Under-abstraction

(false-positives), i.e., they might not correspond to real errors in the system.

The opposite holds for under-abstractions: they contain less behavior than the original system. This is typically a result of abstracting multiple branches into a single step, in effect merging multiple actions leading to different states of the underlying LTS, into a single one. Thus, for safety properties, if there is an error reported in the under-abstraction, this error corresponds to a real error in the original system. On the other hand, the absence of errors in the original system is not guaranteed. *Liveness* properties require that something good eventually happens. While a violation of safety properties can be detected by a finite sequence of transition steps in the model, a violation of liveness properties can only be detected by an infinite execution trace, practically containing a loop which prevents the property from becoming true (i.e., reaching a state in which it holds). If such a loop is present in the under-abstraction, then the reported error is real. Figure 3.6 (right) depicts one way of applying under-abstractions to the vending machine example.

In the following, we use the SMS subsystem as a case study for outlining the established modeling and abstraction guidelines. These guidelines, or rules of thumb, can be applied in a similar manner to other DIRAC subsystems.

3.4.1 Control-Flow Abstractions

Within the mCRL2 toolset, the keyword **proc** is used for defining a named process behavior (called a process variable). This allows for specifying *recursive processes* by referring to process variables within the definition of a process, effectively defining LTSs that have iterative behavior, i.e., loops. As already explained, all agents repeat their implemented logic in iterations: first they read some entries of interest from the service database, then they process the read data, and finally they may write back or update entries, based on decisions from the processing step. Due to this repetitive logic, they can be naturally modeled as recursive processes. Each process represents a distinct agent, specifying its expected behavior when interacting with the service database. Take as an example the code snippet in Listing 3.1, from the *Request Preparation Agent* implementation. Although much of the code is omitted for clarity, the necessary parts for illustrating the basic idea are kept. The first highlighted statement is the selection of all "New" *CacheReplicas* entries. What follows is retrieval of their metadata from a Logical File Catalog (LFC), external to this subsystem. Subsequently, list and dictionary manipulations are done to group the retrieved data depending on the outcome. Two lists of replica IDs are built before the last step: one for the problematic catalog entries, and one for the successful sanity checks. Finally, the last two highlighted code segments update the states of the corresponding *CacheReplicas* to "Failed" and "Waiting" respectively (see Figure 2.9).

Listing 3.1: RequestPreparationAgent.py code excerpt

```
def prepareNewReplicas( self ):
  res = self.getNewReplicas()
  if not res['Value']:
    gLogger.info('There were no New replicas found')
    return res
  [...]
  # Obtain the replicas from the FileCatalog
  res = self._getFileReplicas(fileSizes.keys())
  if not res['OK']:
    return res
  failed.update( res['Value']['Failed'] )
  terminal = res['Value']['ZeroReplicas']
  fileReplicas = res['Value']['Replicas']
  [...]
  replicaMetadata = []
  for lfn, requestedSEs in replicas.items():
    lfnReplicas = fileReplicas[lfn]
    for requestedSE, replicaID in requestedSEs.items():
      if not requestedSE in lfnReplicas.keys():
        terminalReplicaIDs[replicaID] = "LFN not registered at requested SE"
        replicas[lfn].pop(requestedSE)
      else:
        replicaMetadata.append((replicaID, lfnReplicas[requestedSE], fileSizes[lfn]))

  # Update the states of the files in the database
  if terminalReplicaIDs:
    gLogger.info('%s replicas are terminally failed.' % len(terminalReplicaIDs))
    res = self.stagerClient.updateReplicaFailure(terminalReplicaIDs)
  if replicaMetadata:
    gLogger.info('%s replica metadata to be updated.' % len(replicaMetadata))
    # Sets the Status='Waiting' of CacheReplicas records
    #that are OK with catalogue checks
    res = self.stagerClient.updateReplicaInformation(replicaMetadata)
  return S_OK()
```

The logging statements, although critical for operational matters, will not affect the entries' states, and can be ignored (sliced away). Furthermore, instead of tracing back and modeling all variables on which the two final lists depend, we can use non-determinism (and effectively over-abstract) in this particular case. This is because it is not known upfront which branch execution will follow for a particular replica, as it depends on external behavior (i.e., the interactions of the system with its environment). By stubbing out the communication with the LFC and most of the local variable manipulations that follow, and replacing them with a non-deterministic choice between the two ultimate state updates, we can include both possibilities in the model behavior. Of course, depending on the context and the kind of properties we wish to verify, some variable values should not be ignored, in which case determinism can be added using the *if-then-else* mCRL2 statement for

```
mysql> describe CacheReplicas;
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| ReplicaID    | int(11)      | NO   | PRI | NULL    | auto_increment |
| Type         | varchar(32)  | NO   |     | NULL    |                |
| Status       | varchar(32)  | YES  |     | New     |                |
| SE           | varchar(32)  | NO   | PRI | NULL    |                |
| LFN          | varchar(255) | NO   | PRI | NULL    |                |
| PFN          | varchar(255) | YES  |     | NULL    |                |
| Size         | bigint(60)   | YES  |     | 0       |                |
| FileChecksum | varchar(255) | NO   |     | NULL    |                |
| GUID         | varchar(255) | NO   |     | NULL    |                |
| SubmitTime   | datetime     | NO   |     | NULL    |                |
| LastUpdate   | datetime     | YES  |     | NULL    |                |
| Reason       | varchar(255) | YES  |     | NULL    |                |
| Links        | int(11)      | YES  |     | 0       |                |
+--------------+--------------+------+-----+---------+----------------+
```

Figure 3.7: CacheReplicas table description

branching behavior. All relevant selection and update statements are translated into actions parametrized with data. In the following section we will explain how the data is communicated between processes using action synchronization. First, we describe the data abstractions applied.

3.4.2 Data Abstractions

Data abstraction reduces the domains of the variables or objects of the original system into some small abstract domains. The *CacheReplicas* entity contains more information besides the textual status of a file. Every database entry has a unique identifier, descriptive data such as the storage where it resides, its full path, checksum, timestamps, etc., as can be seen from the table description in Figure 3.7. In principle, to make model checking efficient and tractable, it should be performed on finite models, where the domains of all variables are bounded. Since we are only interested in state transitions in terms of changes to the *Status* field, we can collapse most of this descriptive data, and represent this entity as a *user-defined sort* (type) in mCRL2:

$$\begin{aligned} \textbf{sort}\ \mathit{CacheReplicas} = \textbf{struct}\ & \mathit{Start} \mid \mathit{New} \mid \\ & \mathit{Waiting} \mid \mathit{StageSubmitted} \mid \\ & \mathit{Staged} \mid \mathit{Failed} \mid \mathit{Deleted}\ ; \end{aligned}$$

This defines an enumerated data type with all possible status values, as taken from the *CacheReplicas* state machine [2](Figure 2.9). The Tasks entity can be modeled in the same manner. Lists of these sorts can be easily modeled in mCRL2 as *List(Tasks)* and *List(CacheReplicas)*. To define the many-to-many relationship between *Tasks* and *CacheReplicas*, we join these data elements in a tuple:

[2]The missing "Offline" state was added long after this analysis was performed on a model based on the original implementation.

sort *Tuple* = **struct** *p(t:Nat, cr:Nat, link:Bool)*;

The first two elements are the list positions of the *Tasks* and *CacheReplicas* entries, while the last one indicates whether a relation between them exists at a given moment of the system execution.

In reality agents operate on lists of IDs corresponding to the database entries keys, so functions for mapping items of type *List(CacheReplicas)* and *List(Tasks)* to a list of identifiers (i.e., element positions within a list, modeled by natural numbers) *List(Nat)* are necessary. One such *functional data mapping* used in the model is the *tasks2ids* function, which, given an existing list of *Tasks* and a specific Tasks status value, returns the list positions (beginning at zero) matching the value. For example:

$$tasks2ids([Staged, New, Staged, Failed], Staged) \rightarrow [0, 2]$$

The arrow "$\rightarrow$" here represents the result of applying the function *tasks2ids*. The mCRL2 definition of this function is given below.

map *tasks2ids* : *List(Tasks)* $\times$ *Tasks* $\rightarrow$ *List(Nat)*;
map *tasks2ids'* : *List(Tasks)* $\times$ *Tasks* $\times$ *Nat* $\rightarrow$ *List(Nat)*;
var *t, l: List(Tasks)*;
s, a: Tasks;
n: Nat;
eqn $tasks2ids(t,s) = tasks2ids'(t, s, 0)$;
$tasks2ids'([], s, n) = []$;
$(a == s) \rightarrow tasks2ids'(a \triangleright l, s, n) = n \triangleright tasks2ids'(l, s, n+1)$;
$\neg(a == s) \rightarrow tasks2ids'(a \triangleright l, s, n) = tasks2ids'(l, s, n+1)$;

Another example is the *ids2cacheReplicas* function, defined in a similar manner, which can be used to update certain *CacheReplicas* list positions with a new status value, in the following way:

$$ids2cacheReplicas([0, 1],[Waiting, Staged, New], Failed) \rightarrow [Failed, Failed, New]$$

The function will update the first two positions of the original list to a value "Failed". These mappings provide a natural way of modeling the actual database manipulations.

The mCRL2 language does not support global variables (or a similar construct). Therefore, the shared database is modeled as a data wrapper process that maintains the list of data entries as a parameter in its local memory ((*d:List(CacheReplicas)*)). This recursive process continuously listens and responds to requests from other processes (agents), calling itself again after each such selection or update action, as illustrated below.

proc CacheReplicaMem(d:List(CacheReplicas))=

$\sum_{t:CacheReplicas}$ *RPAgent_selectCacheReplicas_rcv(cacheReplicas2ids(d, t), t)*
.CacheReplicaMem(d)
+
$\sum_{l:List(Nat),\ t:CacheReplicas}$ *RPAgent_prepareNewReplicas_rcv(l, t)*
.CacheReplicaMem(ids2cacheReplicas(l, d, t))
+ ...
+
$\sum_{l:List(Nat),\ t:CacheReplicas}$ *RFAgent_removeReplicas_rcv(l, t)*
.CacheReplicaMem(ids2cacheReplicas(l, d, t));

The information exchange is arranged by synchronizing the respective matching actions $\langle Agent \rangle\_\langle action \rangle\_snd$ and $\langle Agent \rangle\_\langle action \rangle\_rcv$ of the processes that are enforced to communicate. The summation operator allows the process to accept any value of the *CacheReplicas* and *List(Nat)* sorts passed via matching actions by the agents. To ensure that such synchronous communication between the data wrapper process and the agents is possible, the mCRL2 *allow* (∇) and *comm* (Γ) constructs are used in the initialization part of the model specification:

∇ {*RPAgent_selectCacheReplicas, RPAgent_prepareNewReplicas,...*
..., RFAgent_removeReplicas}
(Γ {*RPAgent_selectCacheReplicas_snd*|*RPAgent_selectCacheReplicas_rcv*
$\rightarrow$*RPAgent_selectCacheReplicas*,
RPAgent_prepareNewReplicas_snd|*RPAgent_prepareNewReplicas_rcv*
$\rightarrow$*RPAgent_prepareNewReplicas,...*,
RFAgent_removeReplicas_snd|*RFAgent_removeReplicas_rcv*
$\rightarrow$*RFAgent_removeReplicas*})

This allows only those multi-actions that are a result of successful communications between agents and the data wrapper process, to occur. To complete the picture, we give an outline of the model for the *Request Preparation Agent* process:

RPAgent= $\sum_{cacheReplicas:List(Nat)}$ *RPAgent_selectCacheReplicas_snd(cacheReplicas, New).*
(
(*cacheReplicas*≠[])$\rightarrow$ (*RPAgent_prepareNewReplicas_snd(cacheReplicas ,Failed).*
$\sum_{tasks:List(Nat)}$ *RPAgent_selectTaskReplicas_snd(tasks, cacheReplicas).*
((*tasks*≠[])$\rightarrow$*RPAgent_prepareNewReplicasT_snd(tasks, tFailed)* $\diamond$ τ)
+
RPAgent_prepareNewReplicas_snd(cacheReplicas, Waiting)
)
$\diamond$ τ)
.RPAgent;

The translated process is a sequence of different actions for communicating with the *CacheReplicasMem* process. These include obtaining and modifying the data that this process maintains. The agent first selects all the *CacheReplicas* with a status "New". If the selected list is non-empty, the entries are either updated to "Failed" along with their respective *Tasks*, or alternatively updated to a status "Waiting". Since the DIRAC services are merely front-ends for interactions with the database, they essentially act as wrappers around it. We already model the storage as a front-end wrapper process maintaining the database entities in memory, so there is no need to model services explicitly. Finally, the model is put together as a parallel composition of all communicating processes, in the *init* part of the specification.

init RPAgent || SRAgent || ... || CacheReplicaMem([]);

To protect them from growing (possibly) infinitely, the lists of *Tasks* and *CacheReplicas* are initialized and filled only once (using hard-coded values), inside a process modeled following the *setRequest* method implementation (see Figure 2.7). Although the WMS model is larger, it is constructed using similar abstraction rules of thumb. The complete models are available at [4].

3.5 Analysis and Issues

The operation of a model checker closely resembles graph traversal. Typically, model checkers examine all reachable states and execution paths in a systematic and fully automated manner, to check if a certain property holds. In case of violation of the examined property, a counter-example can often be provided as a precise trace in the model, showing which interleavings of actions of the components led to the violation. After the model is written, the state space can be explicitly generated and stored. The state-space generation times and the resulting numbers of states for the two models are presented in Table 3.1. State space generation with the mCRL2 toolset (release July 2011) was carried out on a 64bit Intel Core™2 Duo (1.6GHz) machine with 2GB RAM. This can be a time consuming process, depending on the number of parallel processes, as well as the size of the data domain in the model.

System	Components	States	mCRL2 LoC	Python LoC	Generation time
SMS	4 agents	18,417	432	2,560	<10 sec.
WMS	10 agents	160,148,696	663	15,042	∼12 hr.

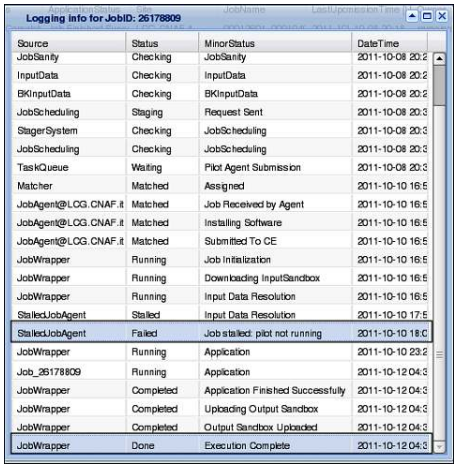
Table 3.1: mCRL2 models statistics [3]

[3]Here we limited the input to one task and two staging files in the SMS, and one job in the WMS.

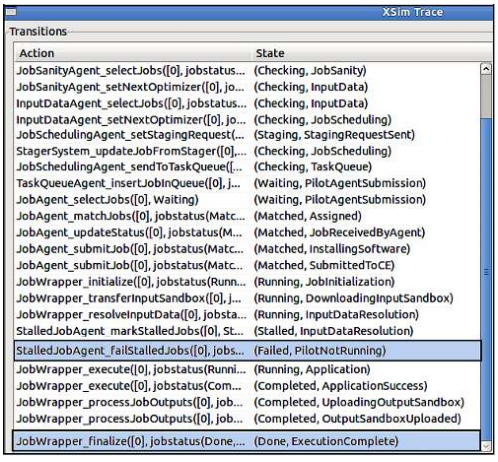
In this particular case, even though the communication between the agents and the shared databases is mostly synchronous, there is non-determinism present due to quantification (sum) over the data domains, at many points in both models. The data domain of the WMS is relatively big, and this makes the state space generation rather involved.

3.5.1 Simulation and debugging

Apart from verification, the mCRL2 toolset has a rich set of tools for analysis of the modeled system. The XSim simulator allows to replay scenarios and inspect in detail the current state and all possible transitions in the model, for every execution step. Modeling errors are not uncommon, especially in the first (draft) versions of the model, and XSim provides the necessary feedback, acting as a debugger. This process was already valuable for understanding the system and identifying interesting behaviors that were later included in the requirements. We want to stress that this is especially useful when building models of existing implementations, where at first glance it is not very clear which (un)desired properties need to be formulated and automatically probed, before they actually surface in the real system. One such problematic behavior was recently reported in DIRAC's production (Figure 3.8a), where a job had transited between two different terminating states, "Failed" and "Done". Replaying the behavior with XSim revealed the exact same trace in the WMS model (Figure 3.8b). This was a result of several circumstances which are difficult to reproduce in the running production and certification setups. The *JobWrapper* process was continuously trying to access a file already migrated to



(a): Logging info of a DIRAC job



(b): XSim simulator trace

Figure 3.8: Invalid job state transitions

tape. Meanwhile, the *Stalled Job Agent* responsible for monitoring the pilot declared the job as "Stalled", and ultimately as non-responsive ("Failed"), since its child process was busy with data access attempt for a long time. However, once data access succeeded, the *JobWrapper* finally started executing the actual payload and reported a "Running" status. The *JobWrapper* assumes that nothing has happened to the status of a job once it brings it to a "Running" state, and only reports a different *MinorStatus* value afterwards. The logic is such because it is naturally expected that this process has exclusive control over the rest of the job workflow, once it starts running. Fixing such a problem without compromising efficiency is not trivial.

3.5.2 Visualization

Reasonably small LTSs can be easily visualized with the interactive GUI tools, as the tools employ smart clustering techniques to reduce the complexity of the image. For instance, the SMS model, with around 18000 states, was visualized with DiaGraphica. Figure 3.9 shows a projection (clustering) of the state space on the *CacheReplicas* memory process, when a single job and one file to stage is given as input in the model, which resulted in only 6 interesting states[4] for this process alone. The graph's nodes contain sets of individual states from the state space, having the same value for the projected state vector element, in this case the list of CacheReplicas. The edges are bundles of transitions connecting these states, with the thickness being proportional to the number of bundled transitions. The transition direction is interpreted clockwise. The clustering allowed us to see precisely which chain of actions by the concurrent agents advanced the state of the *CacheReplicas* entity. In fact, in the modeling phase we discovered the first problematic SMS behavior, blocking the progress of staging *Tasks* (and as a result the progress of jobs). The origin of the

[4]Each state is actually a list with a single element - the status value; for clarity, the list notation and the self-loops are omitted from the figure.

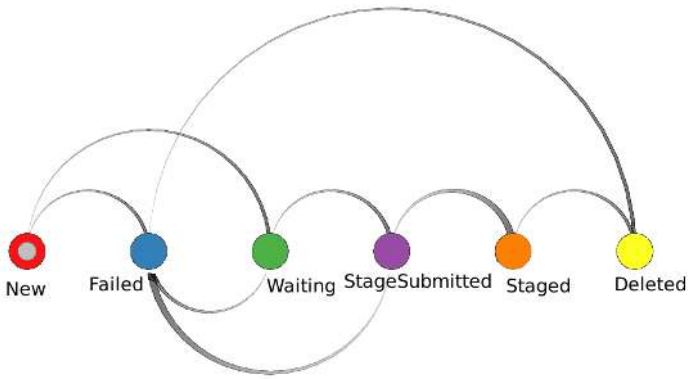


Figure 3.9: CacheReplicaMem process visualization with DiaGraphica

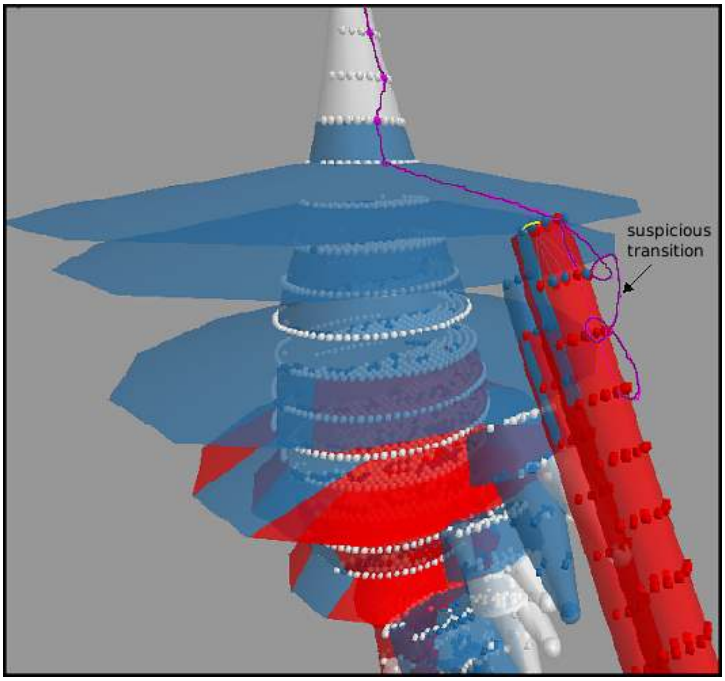


Figure 3.10: State-space visualisation of the SMS with ltsview

deadlock had been a trivial human logic flaw introduced in coding. At a particular circumstance, when a "New" task enters the system **and** all associated replicas are already "Staged" (possibly by other jobs requiring the same input files), its status would immediately be promoted to a wrongly-named (non-existent by design) status "Done". Such tasks would never be picked up by SMS and called-back to the appropriate job, since the agent responsible for this final step would only look for "Staged" tasks in its implementation. A proper state machine implementation instead of hardcoding states in SQL queries can prevent such bugs.

The ltsview tool can be used for a more sophisticated way of visualizing model state spaces in 3D. It also employs clustering of states based on structural properties, but serves to further explore symmetry in the behavior of the system and discover unexpected visual anomalies. The state space of the SMS model is partly visualized in Figure 3.10. The behavior starts at the top of the cluster and proceeds downwards. Branching points indicate alternatives in the behavior of the system. It is possible to mark and color different regions based on values of the state vectors, as well as transitions based on their labels. The particular figure shows a suspicious transition between "Failed" and "Staged" tasks presented by distinct coloring of the regions. One can also interactively simulate the behavior while keeping track of the previous state, the current state, and the possible immediate transitions.

3.5.3 Model checking

The default requirement specification language in the mCRL2 toolset is the modal μ-calculus [85], extended with regular expressions and data. We only introduce μ-calculus informally in this section, and come back with a more detailed syntax and semantics description in Chapter 5. For our current purposes, the regular expressions suffice. These typically allow for specifying that certain behaviors must be absent or present, and *if-then* scenarios. Regular expressions are constructed using constants *true* and *false* and the modalities "[]" and "$\langle\rangle$". The constants *true* and *false* have their usual meaning: *true* holds in every state, while *false* does not hold in any state of the model. The modalities are used for expressing *necessity* ("[]") and *possibly* ("$\langle\rangle$") statements in a formula. *Necessity* means that the formula should hold for every behavior within the brackets, while *possibly* means that there exists a behavior where the formula is satisfied. The behaviors inside the modalities can be specified concisely using *action formulas*. These can be constructed using actions and the operations "||" for union and "!" for the set complement; the action formula *true* represents the set of *all* actions. Any action or action formula can be turned into a set of behaviors using the "$*$" operator, which is used for expressing cardinality, meaning that actions can occur any number of times; the "." operator allows for concatenating behaviors. The last two operators give rise to so-called *regular formulas* in μ-calculus. For example, $\langle a^* \rangle true$ expresses that any sequence of a actions is possible. To express that it should not be possible to do two consecutive *lock* actions, without an *unlock* action in between, the formula $[true^*.\ lock.\ (!\ unlock)^*.\ lock]\ false$ is used.

In mCRL2, the model checking problem is often converted to an equation system solving problem. Such Parametrized Boolean Equation Systems (PBES) can be manipulated and solved on the fly with the toolset, thereby answering the encoded model checking problem. It is worth noting that, even though the generated counterexamples may be somewhat cryptic and difficult to interpret for novices, given that they are shown as trees of PBES variables instantiations, these PBES variables often closely match the data parameters of the original process specifications.

Typical expressions commonly found in practical property specifications with μ-calculus are of the form "$[A]\ false$", which holds when a system has no behaviors matching A, and "$[A]\langle B\rangle true$", which holds if a system has a behavior matching B whenever a behavior matching A has occurred. Using the gained understanding of the system behavior, as well as the reported and anticipated problems, we formulated several requirements for the model checker to systematically probe. In the formulas below, $state(\_\_)$ is an action added to the model to expose states of the entities of interest, augmented as an always-enabled action in the data wrapper process. It should be noted that adding such an action may affect the validity of

certain properties, as the resulting state space will contain a self-loop with the transition $state(\_\_)$ enabled in every state. For example, it will obviously mask potential deadlocks which may exist the original model, since the augmented one can always make progress at least through these self-loops. However, the properties below are preserved under this model augmentation.

1. Each task in a terminating state ("Failed" or "Staged") is eventually removed from the system.

$$[true^*.\ (state([tStaged])\ ||\ state([tFailed]))\ .\ (!(state([tDeleted])))^*]$$
$$\langle(!(state([tDeleted]))^*.\ (state([tDeleted]))\rangle true$$

---

2. A deleted task will never be referenced for transition to any other state.

$$[true^*.\ state([tDeleted]).\ true^*.(state([tNew])\ ||\ state([tStageSubmitted])$$
$$||\ state([tStaged])\ ||\ state([tFailed]))]\ false$$

---

3. Once a job has been killed, it cannot resurrect and start running.

$$[true^*.\ state([jobstatus(Killed, MarkedForTermination)]).\ true^*$$
$$.\ state([jobstatus(Running, JobInitialization)])]\ false$$

Both requirements 1 and 2 were found to be violated, as can be seen from the traces (Figure 3.11). The race conditions manifest themselves in a few subtle interactions between the SMS storage and the agents. In both cases, at step 4 the *Stage Request Agent* issues prestage requests and as a result the CacheReplicas entries (second element of the State column) are updated to "StageSubmitted". Between the moment that this agent has selected the corresponding Task to update to the same state, to the point where the update is done (last step in both traces), other agents may have monitored these replica records and updated them to a new state, along with the associated Task. In practice, the manifestation of such race conditions depends on the speed of the agents propagating the state changes between the selection and update done by the *Stage Request Agent*. Such cases are nevertheless encountered in reality, when this agent has large lists of records to process. This results in a deadlock situation, as the particular Task will have no further state updates made by other agents, since from their perspective its state update propagation is finished. The traces helped to identify the root cause of the problem, and

in both cases, the solution was to improve the implementation such that the Tasks state machine adheres to the original design intentions. In such case, a "Failed" state is a final status and cannot be promoted to a different one.

Model checking and debugging with explicit state space generation was not a viable option for the WMS, due to its size (13 concurrent processes giving rise to a state space of over 150 million states). We therefore resorted to using the LTSmin symbolic reachability tool [29] and the symbolic equation system solver built on top of LTSmin's equation system explorer [116]. LTSmin can also be used as a backend for state space exploration and verification of models written in mCRL2. These tools can reduce the verification times significantly, taking less than 20 seconds for exploring the WMS model (release ltsmin-1.7). They rely on the use of clever data structures such as BDDs [38] for concisely representing the underlying LTS or equation system that is being explored.

The symbolic equation system explorer and solver effectively solves the model checking problems for our requirements. However, in case of violations of the requirements, the symbolic verification tools currently do not provide useful counterexamples. In order to understand the violations, we employed a different trick. We used the symbolic state space exploration tool's option for tracking the occurrence of a certain (specified) action and reporting a trace if such an action is encountered during exploration. For this purpose, we extended the specification with a monitor-

#	Action	State
1	RPAgent_selectCacheReplicas([0, 1], New)	[tNew], [New, New]
2	RPAgent_prepareNewReplicas([0, 1], Waiting)	[tNew], [Waiting, Waiting]
3	SRAgent_selectCacheReplicas([0, 1], Waiting)	[tNew], [Waiting, Waiting]
4	SRAgent_issuePrestageRequests([0, 1], StageSubmitted)	[tNew], [StageSubmitted, StageSubmit
5	SMAgent_selectCacheReplicas([0, 1], StageSubmitted)	[tNew], [StageSubmitted, StageSubmit
6	SRAgent_selectTaskReplicas([0, 0], [0, 1])	[tNew], [StageSubmitted, StageSubmit
7	SMAgent_monitorStageRequests([0, 1], Failed)	[tNew], [Failed, Failed]
8	SMAgent_selectTaskReplicas([0, 0], [0, 1])	[tNew], [Failed, Failed]
9	SMAgent_monitorStageRequestsT([0, 0], tFailed)	[tFailed], [Failed, Failed]
10	SRAgent_issuePrestageRequestsT([0, 0], tStageSubmitted)	[tStageSubmitted], [Failed, Failed]

#	Action	State
1	RPAgent_selectCacheReplicas([0, 1], New)	[tNew], [New, New]
2	RPAgent_prepareNewReplicas([0, 1], Waiting)	[tNew], [Waiting, Waiting]
3	SRAgent_selectCacheReplicas([0, 1], Waiting)	[tNew], [Waiting, Waiting]
4	SRAgent_issuePrestageRequests([0, 1], StageSubmitted)	[tNew], [StageSubmitted, StageSubmitted]
5	SMAgent_selectCacheReplicas([0, 1], StageSubmitted)	[tNew], [StageSubmitted, StageSubmitted]
6	SMAgent_monitorStageRequests([0, 1], Failed)	[tNew], [Failed, Failed]
7	SRAgent_selectTaskReplicas([0, 0], [0, 1])	[tNew], [Failed, Failed]
8	SMAgent_selectTaskReplicas([0, 0], [0, 1])	[tNew], [Failed, Failed]
9	SMAgent_monitorStageRequestsT([0, 0], tFailed)	[tFailed], [Failed, Failed]
10	RFAgent_selectTasks([0], tFailed)	[tFailed], [Failed, Failed]
11	RFAgent_clearFailedTasksT([0], tDeleted)	[tDeleted], [Failed, Failed]
12	SRAgent_issuePrestageRequestsT([0, 0], tStageSubmitted)	[tStageSubmitted], [Failed, Failed]

Figure 3.11: Violation of requirements 1 (top) and 2 (bottom)

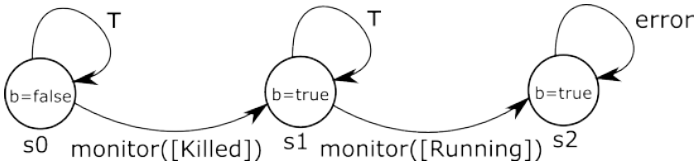


Figure 3.12: Monitor automaton for requirement 3

ing process which fires an *error* action if requirement 3 is violated. This monitoring process is set to run in parallel with the original model, and observes the relevant actions and state changes that the system itself takes. In other words, it accepts exactly those executions that violate the property. As such, it serves as a mechanism for runtime verification, or lightweight bug-hunting. A generic monitoring process for checking μ-calculus properties with regular expressions of the form:

$[true^*.\ state(s1).\ true^*.\ state(s2)]\ false$

---

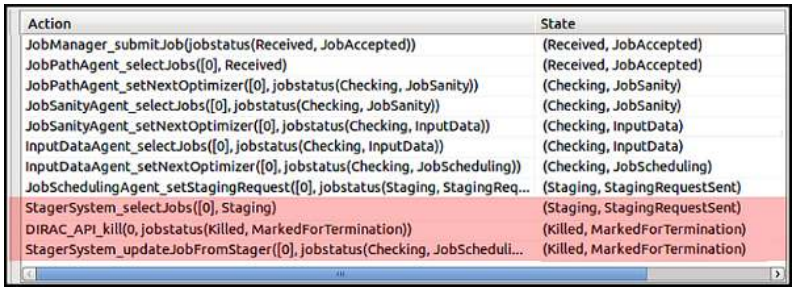
is shown below.

$$
\begin{aligned}
&proc\ Monitor(s1{:}List(Job), b{:}Bool) = \\
&\tau.Monitor(s1,s2,b) \\
&+\ (!b \rightarrow (\textstyle\sum_{s:List(Job)}.(s{==}s1) \rightarrow (monitor(s).Monitor(s1,s2,true)) \\
&\qquad \diamond\ (monitor(s).Monitor(s1,s2,b))) \\
&\quad \diamond\ (\textstyle\sum_{s:List(Job)}.(s{==}s2) \rightarrow (monitor(s).error(s1,s2)) \\
&\qquad \diamond\ (monitor(s).Monitor(s1,s2,b)) \\
&\ );
\end{aligned}
$$

The automaton of the concrete monitor instance which accepts requirement 3 trace is shown in Figure 3.12. Whenever the observed system changes the state of a

Action	State
JobManager_submitJob(jobstatus(Received, JobAccepted))	(Received, JobAccepted)
JobPathAgent_selectJobs([0], Received)	(Received, JobAccepted)
JobPathAgent_setNextOptimizer([0], jobstatus(Checking, JobSanity))	(Checking, JobSanity)
JobSanityAgent_selectJobs([0], jobstatus(Checking, JobSanity))	(Checking, JobSanity)
JobSanityAgent_setNextOptimizer([0], jobstatus(Checking, InputData))	(Checking, InputData)
InputDataAgent_selectJobs([0], jobstatus(Checking, InputData))	(Checking, InputData)
InputDataAgent_setNextOptimizer([0], jobstatus(Checking, JobScheduling))	(Checking, JobScheduling)
JobSchedulingAgent_setStagingRequest([0], jobstatus(Staging, StagingReq...	(Staging, StagingRequestSent)
StagerSystem_selectJobs([0], Staging)	(Staging, StagingRequestSent)
DIRAC_API_kill(0, jobstatus(Killed, MarkedForTermination))	(Killed, MarkedForTermination)
StagerSystem_updateJobFromStager([0], jobstatus(Checking, JobScheduli...	(Checking, JobScheduling)
JobSchedulingAgent_sendToTaskQueue([0], jobstatus(Checking, TaskQueue))	(Checking, TaskQueue)
TaskQueueAgent_insertJobInQueue([0], jobstatus(Waiting, PilotAgentSubm...	(Waiting, PilotAgentSubmission)
JobAgent_selectJobs([0], Waiting)	(Waiting, PilotAgentSubmission)
JobAgent_matchJobs([0], jobstatus(Matched, Assigned))	(Matched, Assigned)
JobAgent_updateStatus([0], jobstatus(Matched, JobReceivedByAgent))	(Matched, JobReceivedByAgent)
JobAgent_checkInstallSoftware([0], jobstatus(Matched, InstallingSoftware))	(Matched, InstallingSoftware)
JobAgent_submitJob([0], jobstatus(Matched, SubmittedToCE))	(Matched, SubmittedToCE)
JobWrapper_initialize([0], jobstatus(Running, JobInitialization))	(Running, JobInitialization)

Figure 3.13: "Zombie" job starts running after being killed



Action	State
JobManager_submitJob(jobstatus(Received, JobAccepted))	(Received, JobAccepted)
JobPathAgent_selectJobs([0], Received)	(Received, JobAccepted)
JobPathAgent_setNextOptimizer([0], jobstatus(Checking, JobSanity))	(Checking, JobSanity)
JobSanityAgent_selectJobs([0], jobstatus(Checking, JobSanity))	(Checking, JobSanity)
JobSanityAgent_setNextOptimizer([0], jobstatus(Checking, InputData))	(Checking, InputData)
InputDataAgent_selectJobs([0], jobstatus(Checking, InputData))	(Checking, InputData)
InputDataAgent_setNextOptimizer([0], jobstatus(Checking, JobScheduling))	(Checking, JobScheduling)
JobSchedulingAgent_setStagingRequest([0], jobstatus(Staging, StagingReq...	(Staging, StagingRequestSent)
StagerSystem_selectJobs([0], Staging)	(Staging, StagingRequestSent)
DIRAC_API_kill(0, jobstatus(Killed, MarkedForTermination))	(Killed, MarkedForTermination)
StagerSystem_updateJobFromStager([0], jobstatus(Checking, JobScheduli...	(Killed, MarkedForTermination)

Figure 3.14: Race condition involving the stager callback no longer occurs

job to "Killed", the automaton also changes its state to $s1$ by enforcing synchronous communication between the respective actions $monitor(\_)$ and $state(\_)$. The silent step (τ) can be executed as a result of any non-relevant system state change from the perspective of the automaton. As soon as the system changes the state of the job further to "Running", the automaton makes a transition in a similar fashion, after which point the action *error* is executed, and can be detected by the LTSmin tool. If such a full cycle is accepted by the automaton, the temporal property is indeed violated. Using this technique, a counter-example (Figure 3.13) showed that, when staging was involved in the workflow of a job, the callback from the SMS was not properly handled. It awakened the job immediately to a "Checking" (and eventually "Running") state even if it had been manually "Killed" (using DIRAC's command line API) by production managers meanwhile. Such "zombie" jobs were discovered in the system on several occasions, and with the model at hand it was easy to replay the behavior and localize the problem.

The implementation was fixed to properly guard the callback against this particular case. The changes were also reflected back to the mCRL2 model [4]. Figure 3.14 shows that the stager callback no longer affects the job state if it has been changed from its original "Staging" one in the meantime. The state space of the fixed model amounts to 157,048,664 states, the reduction being a result of removing the incorrect behavior. However, we must note that model checking still discovered traces where the property is violated, even without the stager being involved. The current architecture does not permit an easy workaround for this problem. Protecting shared memory access using traditional locking techniques does not scale well. At the time of writing, there was an ongoing effort in redesigning DIRAC's job workflow, to accommodate a task dispatcher that would take care of persisting the state of the tasks to the storage backend, and distributing them among so called "executors". This should replace the old agent components' polling architecture, and reduce the load they place on the storage, in addition to eliminating race conditions. We will come back to DIRAC's Executor Framework in Chapter 4.

3.6 Discussion and Evaluation of the Approach

The author of this dissertation had no previous experience with process algebras, before embarking on the quest of formal modeling with mCRL2. Although it is difficult to estimate the total effort, combined with other research activities, both subsystems were modeled incrementally and the problems were found within a few months; this is including the time it took to analyze and understand the original implementation being modeled. None of the bugs discovered with this formal approach were localized and fixed by DIRAC developers within this timespan, although the manifestation of the faulty behaviors was already evident before the subsystems were modeled. Analysis of this kind is typically performed after a problem has already surfaced in the real system, as a means to understand the events which led to it and test for possible solutions. We managed to come across an actual bug at the same time it was observed in practice, which increased our confidence in the soundness of the model. As a positive side effect of having the formal models at hand, the author gained a much greater insight into the emergent behavior of these subsystems, in a manner impossible for distributed systems, if based solely on running the implementation and analyzing logging traces. Concurrency bugs are often a result of unanticipated interleaving of events, and there is almost no control over their temporal ordering while the distributed DIRAC agents and services are autonomously running. With the simulation tool we had a full control over stepping through the possible executions, at a pace which allowed to understand the dynamics, while having the full execution history trace at a glance. Some of the more challenging aspects were:

- Making the mind-switch to the process algebra world, starting from a high-level object-oriented implementation, where a process execution viewpoint is not so obvious, but rather the focus is on the behavioral interface that the objects provide, and the communication between them. Once the initial resistance was overcome, it turned out that mCRL2 is indeed intuitive and natural enough to express these aspects.

- Interpreting the mCRL2 counter-examples that are shown as trees of PBES variables instantiations. This requires some experience and ingenuity, as they cannot be directly fed into the simulator for replaying the behavior. As a rule of thumb, using the monitor approach whenever possible, in order to turn the model checking problem into a reachability (of an *error* action) one, provides much more user-friendly traces.

- Choosing the right abstraction level: in the choice of which implementation details to discard, we were guided by the existing reported problems. However, if problems of a different nature are to be checked, this may require some

modifications in the models. Furthermore, due to the use of non-determinism (over-abstraction) in both models, model checking requirements 2 and 3 produced false-positives. The decision of discarding such counter-examples must be made in consultation with the original implementation.

- Deciding which properties to be checked: while we formulated only a few basic properties based on the reported problems within the collaboration, we conjecture that many other unexpected and undesired behaviors still exist, and it is only a matter of time before they surface in the running system.
- Using the right combination of switches available in the tools to make the analysis more efficient when the resources for performing model checking are scarce. Consulting mCRL2 experts helped tremendously with this.
- The models can quickly become outdated, and keeping them in sync requires a continuous effort. DIRAC evolves on a daily basis, whether it concerns small bug patches, or more involved developments, releases are made weekly.
- Dealing with state space explosion: as a rule of thumb, when the state space generation grows suspiciously large without prospects of completing within a reasonable time, it is worth looking into data structures (like lists) potentially growing without bounds, processes not being set to correctly synchronize and communicate on actions like intended to (asynchronous behavior can increase the exploration time significantly). Isolating problems by incrementally hiding parts of the behavior (with the available mCRL2 mechanism for this) in order to localize the bottlenecks, showed to be a useful strategy. Modular or compositional verification [23] is a powerful abstraction technique that allows to concentrate on the relevant components for the specific property we wish to check, abstracting from irrelevant ones. However, for modular verification to be effective in reducing the state space, the system properties must be decomposable into properties over the individual components, which is often far from trivial to achieve.
- Loosening the tension between applying formal methods to an existing large system and getting the detected problems resolved: some of the discovered bugs are not easily fixed, for instance, due to performance impacts or architectural restrictions. It seems like efficiency considerations in a distributed system design have a higher precedence than correctness ones. An inefficient system can be unusable, whereas a buggy program can still be used (granted, with some frustration). When the system architecture is already in large part established and implemented, the resistance of applying bug fixes which may impact the architecture, tends to be more pronounced. This resistance depends on several factors: on one hand, the estimated effort needed to adapt

the design, the number of people who have the time, knowledge, and the will to estimate and implement the change, and on the other hand, the expected nuisance it will cause to operations. Over time these factors change, especially in small dynamic teams like the one behind the development of DIRAC. If the formal model (rather than the implementation) would be the first one affected by the modification, the number of new problems introduced should decrease quicker over the lifetime of the system, since problems are caught earlier. Deeper understanding of the cause of a misbehaving system, rather than just the symptom, leads to different conclusions about the necessary system "patch". On the other hand, formal methods do not (yet) propose a direct solution to the problems that they can discover.

3.7 Related Work

The application of model checking in communication protocols [104; 16; 145], safety-critical software [194; 111; 138], driver design [18; 30], among others, has demonstrated the maturity and potential of tools for automating such formal analysis. Case studies in the domain of high-performance distributed computing, and in particular concurrent/MPI-based program design, have surfaced as well [178; 139; 176], mostly relying on the MPI extension of the popular model checker SPIN [177]. The input language of SPIN is a state-based, imperative one. Specifications are described as concurrent communicating state machines (processes), based on Dijkstra's guarded commands notation with "goto" statements. Compared to mCRL2, SPIN has more limited language capabilities when it comes to describing custom data types. Furthermore, reasoning in terms of state machines, with a focus on how each individual object may possibly jump through its states, rather than in terms of scenarios and actions, had shown to be less natural for systems like DIRAC, in our initial modeling attempts with SPIN.

To overcome the problem of state space explosion, several variants to explicit model checking have gained popularity. Runtime verification [52; 103] aims to increase confidence in the correctness, but does not make claims for absence of defects in a system. The target program is instrumented manually, or automatically by scanning function calls and variable assignments, in order to capture and emit relevant events which are then monitored by a special component during execution. This component resembles our monitoring process automaton used to cope with state space explosion during model checking of DIRAC's WMS system. While runtime verification is useful for capturing incorrect behavior of the system implementation, code instrumentation may not always be a feasible option, considering the increase of memory consumption. Furthermore, runtime monitor instrumentation becomes very challenging for distributed systems, where there is no global clock for reference

and ordering of events.

Similar to this lightweight form of verification is static program analysis [173]. Rather than code instrumentation and execution, the objective of static program analysis is to determine runtime properties of systems at compilation time. A special analyzer targeted for the implementation language maintains information on the control flow and function call graphs of the software. Approximation is used by such tools in practice, in order to improve scalability. The obvious advantage of this approach is that it works directly on the implementation. However, most static analysis tools aim at discovering more general and low-level implementation bugs, such as uninitialized variables, null pointer references, zero division, or out of bound array indices. It is almost impossible for a general static analysis tool to figure out the exact high-level intent of the implementation code taken as input. Because of the heavy approximation they use, static analysis tools are quite fast and applicable to large software projects, but result in many false negatives reported. An additional drawback is their practical availability for languages other than C and Java. Both runtime verification and static analysis can be seen as additions to traditional model checking, rather than alternatives. Some progress has been made in the direction of combining the approaches [194; 30; 34]. For instance, Java PathFinder [194] relies on static analysis to reduce the state space, before systematically exploring it by executing the abstracted program. In [34] static analysis and model checking also operate in a feedback loop.

Static analysis is, in a sense, an extreme form of automated deductive verification. In its original form, deductive verification technology allowed for the use of logics such as Floyd-Hoare logic [105] for showing the correctness of a software or hardware system by constructing proofs thereof, based on mathematical descriptions of the behavior of the system. These days, the proofs involved are often discharged using a mechanical or an automated theorem prover. Moreover, more advanced logics such as Separation Logic [168] are used. Deductive verification is applied quite successfully on software [149; 118] and hardware [10], but the effort required for conducting the verification can be prohibiting, and can compare poorly to model checking.

Instead of constructing a model of the system from scratch, a technique called process (or workflow) mining [8] can be used. It is a method of automatically distilling a process or software specification from a set of structured traces collected from real system executions. The mined specifications are typically expressed in a form of graphical formalisms such as Message Sequence Charts or Petri-Nets. Given that the analysis of program execution traces is still the most prevalent debugging technique for distributed systems, and DIRAC's components produce immense amounts of execution logs anyhow, it seems like an attractive alternative for mining models. There is a vast amount of work in this area [188; 132; 195; 189; 12]. A

common assumption in process mining is that the events in each trace are totally ordered, which for concurrent systems does not always hold. To cope with this, most approaches require that the thread IDs or process IDs are recorded in the traces as well. Observing this as a realistic obstacle for obtaining an accurate model, Lou et al. in [132] propose an algorithm for constructing models from traces of concurrent systems by expressing events as partially ordered. Their algorithm identifies dependency relationships between pairs of events in interleaved traces. Another common assumption is a direct result of the underlying techniques used for constructing the models, namely data mining and machine learning. These techniques consider "erroneous activities" or activities that were not logged in the appropriate order for some reason, as noise, and disregard trace sequences with no statistical significance. As a result, a design flaw can be mistaken for a noise, or not captured at all if a particular execution path has a low probability of occurrence. A notable exception in this direction is [190], where the emphasis is not only on process discovery, but also on verification: desired or undesired properties can be expressed with Linear Temporal Logic (LTL), and verified against the traces.

While relying solely on process mining as a technique for extracting a formal model may not be the way to go, inferring a "fuzzy" model from the frequent interaction traces, rather than manual code analysis could be a helpful initial step. Nevertheless, the graphical models obtained must be translated into a process algebra formalism accepted by an actual toolset, if system correctness is to be established.

3.8 Conclusions

In this chapter we have applied a formal approach for modeling and analyzing CERN's DIRAC grid system. We used two subsystems as case studies: the Workload Management and the Storage Management System (Chapter 2), which are the driving force of DIRAC for production workflows. By creating an abstract model, simulating, visualizing, and model checking it with the mCRL2 toolset, we were able to gain insight into the system behavior and detect critical race conditions which were confirmed to occur in the real system. Referring back to **Research Question 1**, this provides a strong evidence to positively conclude that mCRL2 is suitable and rich enough for modeling and addressing design errors of realistic-scale distributed data-driven systems, such as DIRAC. Furthermore, addressing **Research Question 2**, the advantages of using model checking in this context become apparent. Despite the continuous DIRAC development for over a decade, some of these problems were notoriously difficult to discover and trace. Testing of concurrent systems and reproducing error traces is often challenging because of the lack of control that the tester has over the execution of the concurrent processes. With the model at hand, replaying the traces and localizing the problems became much

more straightforward.

Writing a sound formal model based solely on the existing implementation and scarce descriptive documentation is challenging. The code-base and the number of grid components engaged in providing DIRAC's functionality outnumber many existing model checking use cases. The choice of Python as implementation platform has led to the prevailing usage and manipulation of dynamic structures such as lists and dictionaries, challenging the transition to an abstract formal representation. We believe that the combination of code slicing and the abstraction rules of thumb presented in this chapter can be helpful for modeling such data-driven concurrent systems like DIRAC, where many of the problems arise from sharing data among processes.

The investment in writing an abstract model in a formal language pays off proportionally to the size (number of components) of the distributed system. This holds especially for grid systems where a large number of concurrent components of a similar kind deliver the overall system functionality. However, deriving a formal model should start with a behavioral specification at a higher abstraction level altogether. Furthermore, this derivation should ideally be automated, to avoid modeling mistakes and to systematically keep the formal specification in sync with the continuous system changes.

Chapter 4

Model-Transformation Methodology

4.1 Introduction

As the complexity of modern software systems increases, a major challenge is faced in maintaining their quality and functional correctness. Early discovery of design errors which could potentially lead to deadlocks, race-conditions and other flaws later on in the development phase, is of a paramount importance, as was already indicated in Chapter 3. A common practice in software engineering is to provide designs or models used for communicating and validating the system requirements, *before the system implementation, testing, and deployment phase begins*. The Unified Modeling Language (UML) [90] has become the lingua franca of software engineering, in particular for the domain of object-oriented systems. Over time, several mature CASE (Computer Aided Software Engineering) tools have already adopted UML as the industry-standard visual modeling language for describing software systems. However, the expressiveness of UML sometimes gives rise to detailed models, where emergent behaviors that are not expected by the designers, still manifest. The language of UML is defined by its abstract syntax, its concrete syntax and its semantics. The abstract syntax is described by the UML metamodel [91]. It defines all the valid modeling elements, their attributes and relationships. Their graphical representation, as used when visually modeling systems in practice, comprises the concrete UML syntax. Although useful, the semantics is given in terms of natural English language explanations, which sometimes results in confusion and ambiguities.

To remove these ambiguities, as well as bridge the gap between industry-adopted methodologies based on UML software designs and the formal analysis and verification tools, many approaches have been proposed for automated extraction of the necessary analysis models from the UML artifacts. For instance, Petri Nets [65; 25; 200], Layered Queuing Networks [151] and stochastic process algebras

[183; 184] are used for functional or performance analysis of dynamic UML models augmented with quantitative performance annotations (such as resource demands, timing and probabilities of different execution steps). Model checking for certain properties of the system is often done via translation into process algebra [95; 115]. Automatic synthesis of functional test cases from UML models is possible as well [19; 152]. Model-to-model transformations can also be done within the UML domain itself, for the purpose of model optimization or refactoring [196], or between a UML model and a target formalized programming language subset, using graph transformations [117; 71], to show that the UML behavior coincides with that of the corresponding program.

A UML model of a system is typically a combination of multiple views. Devising an automated model transformation methodology requires that behavioral views of the system be available. The static views of a system (such as Class and Deployment UML diagrams) are rarely sufficient to extract the necessary information for constructing a target model for performing meaningful behavioral analysis. Activity, Sequence, and State Machine diagrams are among the most commonly used behavioral ones for this purpose.

State Machines (SM) represent the reaction of individual objects to external stimuli. They are suitable for describing specific parts of systems, such as a critical control component, but are rarely used as the sole paradigm for developing large distributed object-oriented systems. Software designers almost never create a fully-formed object a-priori and in isolation, with all the behavior that the object will ever need. Typically, state machines are more heavily used in domains like embedded systems, which are control-oriented, rather than data-oriented.

Activity Diagrams (AD), with semantics based on Petri Nets, describe the system at a higher level of abstraction, where objects and message exchanges are not captured, but rather, inter-process control flow is depicted. They represent workflows of activities and actions, with support for choice and concurrency, and are commonly used for business process modeling.

Sequence Diagrams (SD) provide the most fine-grained runtime view of the system. They model a set of interacting objects by means of message-exchanges over time. These diagrams contain information about the control flow during the interaction, capture conditions and iterations, as well as intra-process concurrency. With the introduction of UML 2.x rich set of features such as combined fragments, SDs have become popular for expressing scenarios because of their clear and intuitive visual layout and close correspondence with actual programming code-like structures.

Most of the proposed transformation approaches up to date target only one particular type of behavioral diagram, mostly AD or SM diagrams [130; 66; 183; 115; 39]. When it comes to interactions (SDs) or targeting multiple diagram types, the

existing approaches mostly deal with UML 1.x semantics [184; 171; 159; 61; 154; 172], largely limiting the expressiveness by not taking into account all elements which allow designers to describe complex traces in a compact manner. Furthermore, they mostly deal with communication structures only, focusing on the order and types of messages, and abstract away from custom data types (beyond simple Boolean and integer types) exchanged in these communications, which are not easily expressible in some of the formalisms. Finally, with the exception of [99], there is generally no feedback on the results of the formal analysis, back into the UML domain.

Our interest is obtaining a formal model in the process algebra mCRL2, taking a UML model as starting point. Due to the true-concurrency nature (multi-actions) of mCRL2, we can faithfully express the partial-order semantics and the concurrent behavior of interactions. This is important in distributed systems, where the representation of concurrency by an interleaving interpretation ignores true concurrency resulting from the asynchronous behaviors that may exist. The proposed approach in this chapter is based on the UML 2.x semantics, and makes use of both sequence and activity diagrams, combined with the static class structure of the system, to automatically derive the target formal model. We rely on the textual XMI model representation to devise the model transformation procedure. XMI [90] is an XML-based vendor-independent format for metadata exchange between compliant UML tools. Based on the approach, we have developed a prototype tool that can take a UML model in XMI representation as input, and construct the mCRL2 model. Our methodology allows traces from the model checking tool to be conveniently displayed back in any UML tool. We have further applied the toolset to DIRAC's Executor Framework [157]. Prior to applying the toolset, we produced the dynamic UML views by reverse-engineering the existing code, in consultation with the main developer.

The remainder of this chapter is organized as follows. We start with a brief introduction to the syntax and the informal semantics of sequence and activity diagrams in Section 4.2. Based on these behavioral UML diagrams, in Section 4.3 we present an automated transformation methodology for deriving formal models in mCRL2. Here we discuss the semantic choices we made with respect to some ambiguous points in the semantics as described by the OMG. In Section 4.4 we demonstrate the applicability of the transformation on the newly developed components of the DIRAC WMS module. As we will see, model checking the generated mCRL2 model effectively discovered the root cause of a problem that had been detected during testing of these components, providing some confidence in the correctness and usefulness of the methodology. In Section 4.5 we reflect on the efficiency of the methodology, and our experience on the consequences of using it, and we conclude in Section 4.6.

4.2 Modeling Interactions in UML

To translate a system composed of different diagrams into a formal model, we chose sequence diagrams as a driving behavioral description type, and we take the necessary additional information about inter-process concurrency from activity diagrams. Our choice is motivated by the fact that SDs provide the richest set of constructs for low-level behavior expression, and as such have a close correspondence with actual code. Additional information from ADs is necessary for deriving the actual (OS level) processes, relevant for concurrent and distributed systems. Such details cannot be captured with SDs, as we will see.

4.2.1 Sequence Diagrams

Sequence diagrams model the interaction among a set of participants, with emphasis on the sequence of messages exchanged over time. The participants are class instances (objects) shown as rectangular boxes, with the vertical lines falling from them known as *Lifelines* (Figure 4.1). Each *Message* sent between the lifelines defines a specific act of communication, synchronous or asynchronous. The start and end events of the directed edge representing a message are called *MessageEnds*, and are marked with a so called *MessageOccurrenceSpecification* element of the UML metamodel. Synchronous messages are drawn with a filled arrow-head, while asynchronous ones have an open arrow-head. In Figure 4.1 (left) the message *remoteCall* is asynchronous, so *objC* continues with execution immediately after the call, not waiting for a completion and reply from *objD*. Reply messages are drawn with dashed lines. All message types can carry arguments.

Messages are sent between objects with the aim of invoking specific behavior, known as *ExecutionSpecification*, and visualized as a thin rectangle on the receiver's lifeline. Thus, execution specifications illustrate when the invoked method is executed on a particular object. Execution specifications can be nested/overlapping, as a result of a callback message, or an object invoking its own method, an exam-

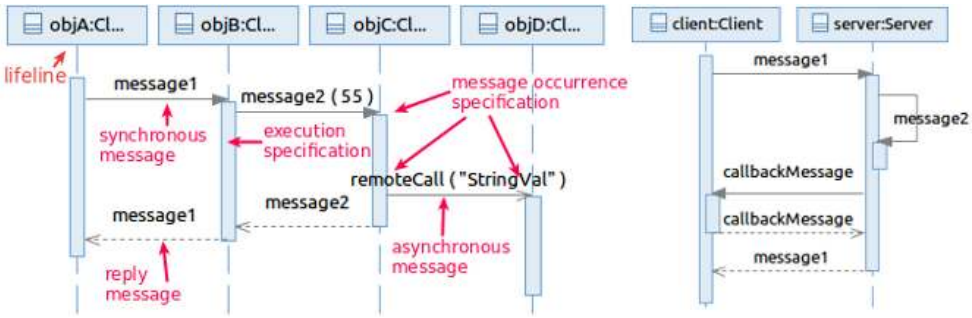


Figure 4.1: Sequence diagrams notation

ple of both situations shown in Figure 4.1 (right). In this example, the *client* object sends a request for *message1* execution on the *server* side, after which the execution is blocked until it receives a reply from that method call. However, this does not stop other potential objects from invoking any method of the *client* and *server* interfaces. This possibility of overlapping (possibly concurrent) method executions on the lifeline on a single object plays an important role in our transformation methodology choices, as we will see shortly. For one, it implies that objects should not be treated as sequential processes, an approach taken often when translating sequence diagrams to process algebra models. Furthermore, with such approaches, in the presence of combined fragments with guards, a local condition evaluation of an object (lifeline) must be visible to all communicating processes in the formal model (using global variables).

Combined Fragments

Basic sequence diagrams capture finite interaction sequences without any branching points. In this "linear view" of time, each moment in the interaction has one possible future. To support branching-time behavior, where each moment has multiple possible futures (either as a result of decision moments, or concurrency), combined fragments were introduced in UML 2.x. They add more expressiveness to SDs by means of constructs capturing complex control flows in a compact manner, thus overcoming many limitations present in UML 1.x [90]. The specification supports different fragment types, such as *alt*, *opt*, *loop*, *break*, *par*. They are visualized as rectangles with a keyword indicating the type. Combined fragments consist of one or more *InteractionOperands*. Depending on the type of the fragment, *InteractionConstraints* can guard each of the interaction operands. The guards values determine which fragment's operand(s) will be executed at runtime. Combined fragments can be nested with an arbitrary nesting depth, to capture complex workflows. Figure 4.2 shows how some of them can be used.

The *alt* fragment indicates an alternative between two or more choices. All operands must have an explicit or implicit guard expression that evaluates to true or false. If an operand has no guard, an implicit true is implied. If none of the operands' guards evaluates to true, none of the operands are executed and the remainder of the enclosing fragment is executed. If multiple operands evaluate to true, one is chosen non-deterministically. In the given example, this fragment has two operands, only one of which is chosen depending on which of the guards evaluates to true. In this case, if $intValue > 0$, *objE* sends the message *message2*, otherwise if $intValue < 0$, it calls its *innerMethod*. The *opt* fragment is semantically equivalent to an *alt* fragment where there is only one interaction operand, and the second one is empty.

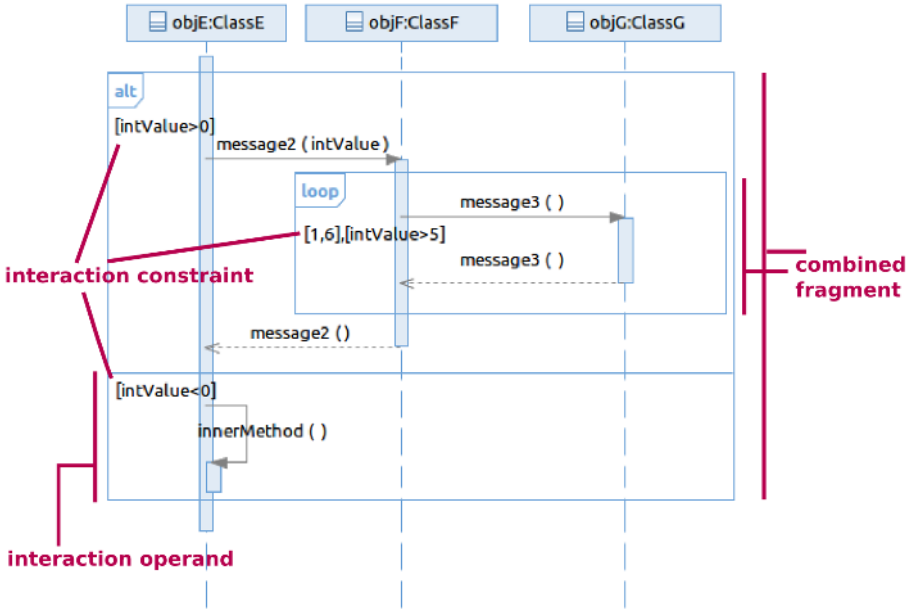


Figure 4.2: Combined Fragments examples

The *loop* fragment models repetitive interactions. The guard may include lower and upper limits on the number of iterations, as well as a Boolean expression. In practice, however, either of these conditions is used, but rarely both in combination. The semantics is such that, after the minimum number of iterations have executed and the Boolean expression is false, the loop will terminate. In the example of Figure 4.2, the loop will be executed 6 times, but only if the Boolean guard $intValue > 5$ evaluates to true. This semantics is ambiguous, so there is room for interpretation, as we will point out in more detail in Section 4.3.2.

The *par* fragment (Figure 4.3 left) models parallel execution of the behaviors in its operands. Its intention is to show intra-process parallelism, i.e., separate threads of execution within a scenario. The operands can be interleaved in any way, as long as the ordering imposed by each operand is preserved. There is implicit synchronization on two events: all operands are started when the fragment is entered, and are stopped when the fragment is exited, before the rest of the scenario (if any) can continue.

The *break* fragment (Figure 4.3 right) captures an exit point, in the sense that the behavior inside the operand is performed instead of the remainder of the enclosing fragment, when the guard evaluates to true. When the guard of the operand is false, the behavior inside it is ignored, and the execution continues in the rest of the enclosing fragment. The *break* combined fragment is much like the **break** keyword in programming languages like C++ or Java.

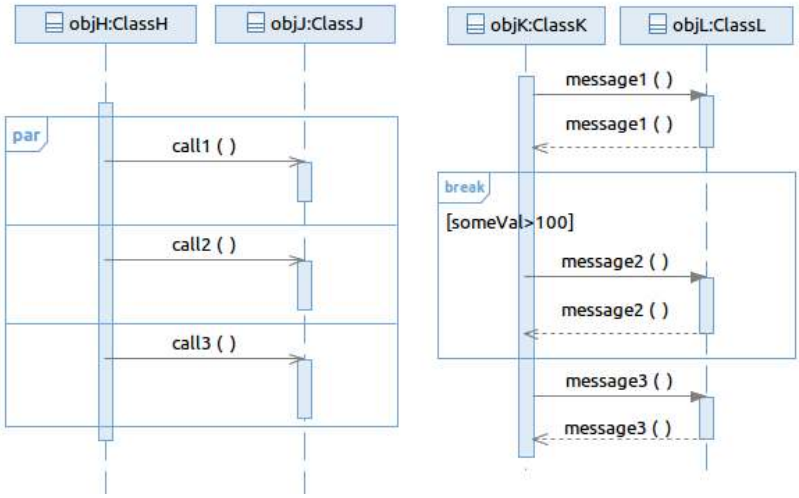


Figure 4.3: Example of par (left) and break (right) fragments

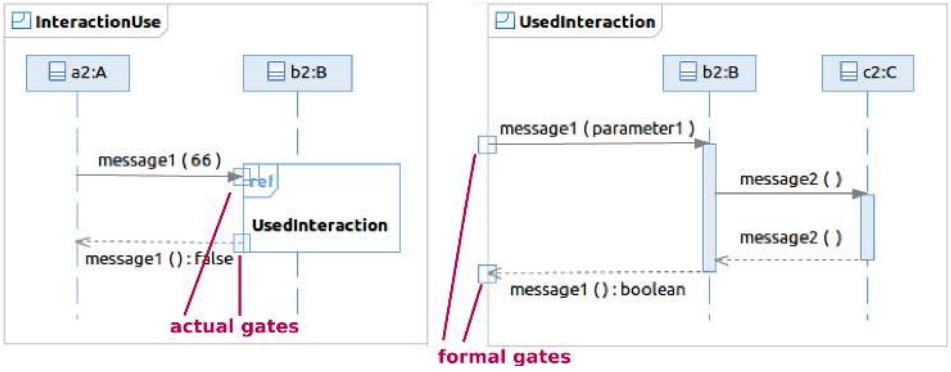


Figure 4.4: Formal and actual gates in InteractionUse

All fragment types described above have equivalent constructs in most (object-oriented) programming languages. Another useful enhancement is the *InteractionUse* fragment. For expressing complex scenarios, one can include a reference to another SD, which is semantically equivalent to including the behavior of the called diagram in situ in the calling one. This promotes reuse of already defined sequence diagrams. To facilitate such behavioral constructs, model elements called *Gates* are used. The referenced diagram has *FormalGates* placed on its boundaries, allowing messages to come to, or from its environment. A matching set of *ActualGates* are placed on the *InteractionUse* box, as shown in Figure 4.4. In that sense, *Gates* are *MessageEnds* which connect the *Messages* inside and outside the referenced diagram, and are added to the UML metamodel to satisfy the syntactic constraint that messages must have two message ends.

Runtime Semantics

Unlike the syntax, the SD semantics is scattered through the UML specification provided by the OMG (Object Management Group), and defined by means of natural English language. The most important points can be summarized as follows: an object sends a message as a result of earlier behavior invocations on its interface, and is the object's reaction to these receptions. Message and execution completion are considered local concepts. For a message *m* sent from object *o1* to object *o2*, the sender's view of that message completion is the sending, the receiver's view of the message completion is its reception, while other objects have no knowledge of *m*. Thus, the only synchronization points between the objects are the message exchanges.

The semantics of an SD is described in terms of a pair of valid and invalid *traces*. A trace is a sequence of events, or *OccurrenceSpecifications*. The orders of events in an SD are defined using *partial order* semantics. This effectively means that the events are only subjects to the following constraints: a send event for a message always occurs before the receive event; and all event occurrences on a particular lifeline are totally ordered. However, there is no further order restriction imposed on unrelated events appearing on different lifelines. All valid traces produced from a diagram must satisfy these order constraints. *This semantics does not impose a total ordering of the messages in a given SD.*

4.2.2 Activity Diagrams

As already stated, we use ADs to extract concurrency information necessary for deriving OS-level processes in a distributed system setup. Although the notion of concurrency is present in some form in SDs, the *par* fragment only indicates that the implementation can execute any interleaving of the operands' behaviors, without mandating that the implementation be concurrent or distributed. In a concurrent or distributed setup, each of the SDs could be parts of multiple processes that must be initialized by the system environment at some point, and this is where elements of ADs come into play. We give here only a limited subset of the concrete syntax necessary for our transformation methodology.

An example of a workflow modeled with this diagram type is illustrated in Figure 4.5. Activity diagrams consist of *Action*, *Activity*, and *Control* nodes, connected by *Control Flow* edges, with each diagram having one distinguished *Initial Node*. The control flow nodes have their intuitive meaning as in traditional flow charts, namely to depict concurrent flows (*Fork*), their eventual join or synchronization (*Join*), and decision points (*Decision*). While there are various action types, we are primarily interested in the *CallBehaviorAction*, which invokes a referenced behavior directly. Since SDs are also classified as behavior, we use this action type to provide the link

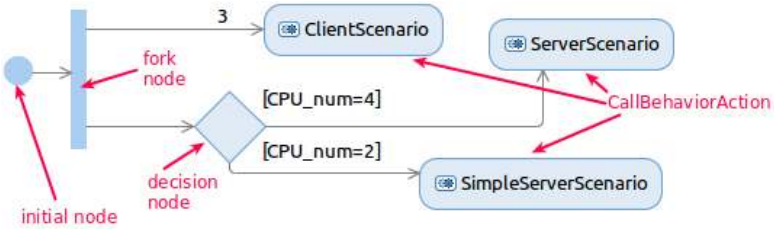


Figure 4.5: Activity diagram example

between SDs and the concurrent system setup described in an AD. An alternative way to model the same overview of control flow is by using *Interaction Overview Diagrams*, though these are less frequently used in practice. They are essentially simplified versions of ADs, where each individual activity is pictured as a frame that can contain a nested interaction diagram. The rest of the notation elements are the same as those of ADs, namely initial, final, decision, merge, fork, and join nodes. The transformation described in this chapter can be easily extended to take these into account as an alternative to ADs.

In the given example, three process instances are initiated with the *ClientScenario* SD, concurrently with an instance of *ServerScenario* or *SimpleServerScenario* (depending on the guards). Since UML 2.x, the token-flow (Petri Net-like) semantics has been applied to ADs, where the rules for control and object flow are defined by using tokens. A token corresponds to an executing thread. The weight on the edge calling *ClientScenario* defines the number of tokens that can flow when the edge is traversed. Multiple SDs can be chained in a sequence or set to run concurrently in a similar manner, allowing more complex workflows to be described.

4.3 Transformation Methodology

4.3.1 The Object-Process Problem

Before describing the transformation methodology, we outline the rationale behind the choices we made, and how they differ from previous approaches that deal with SDs as behavioral description diagrams of a system. The approaches of which we are aware [154; 131; 184; 201], and which use a process algebra formalism for a target model, translate each lifeline (hence, each object) into a sequential process[1]. However, this implies that an object behaves intrinsically sequentially, which is not always the case. The object's individual behavioral capabilities are exposed via its methods. In other words, an object is a component with multiple access points. In a concurrent setting, multiple threads of a process (or even multiple processes sharing an object, if the implementation language permits this) could be invoking

[1]The only exception made to this rule is when dealing with the *par* fragment.

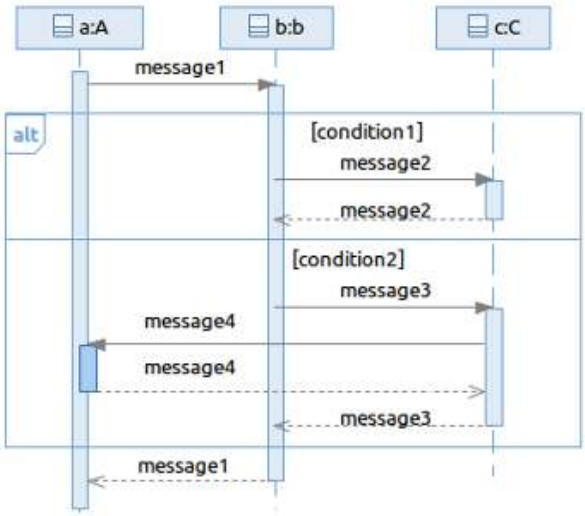


Figure 4.6: Sequence diagram with alternative control flow

methods of the same object, thus, that object could be executing multiple behaviors at the same time. In that sense, behaviors that are requested/offered at the same time should not be excluded.

Consider the simple SD in Figure 4.6. Even if we assume that the scenario is executed by a single OS process, treating each object as a sequential process is problematic. In the example, after invocation of *message1*, object *a* needs to know the choice that object *b* has made (modeled with the *alt* fragment), while this choice is based on local conditions that only *b* is aware of. Therefore, *a* cannot know whether its method *message4* will be invoked by object *c*, before a return from *message1* is received on the same lifeline. Consequently, a single process representation of the lifeline *a* should not control the reception of *message4* and execution of the associated behavior. Some approaches attempt to deal with this by making all involved processes aware of each others' local decisions (using global variables), but this quickly becomes cumbersome and prone to errors, given how complex UML 2.x SDs can be made by nesting combined fragments. Furthermore, such global explicit synchronization of several lifelines is not appropriate for distributed systems, where a guard should only be evaluated by one lifeline.

We wish to preserve the OO paradigm in the transformation to a formal model. In this paradigm, unless an object is active, it does not control the invocation of its methods; it only responds by executing the associated behavior. An OS process is then essentially a chain of method invocations on objects. To achieve this, we associate an mCRL2 process *description* with each class method. A description (be it an actual program code or a UML model) of a class method should not differ across objects that are instances of that class. Of course, at runtime objects execute

only some of the multiple possible traces captured by that description, based on variable values. In our methodology, each such mCRL2 process *instance* carries data parameters that encode the class, object, and OS process instance to which the exhibited method behavior belongs at runtime. As an important consequence of this choice, *we preserve information on objects, classes and method calls in the mCRL2 model*, which makes it easy to reverse model checking counter-example traces back into the UML domain.

4.3.2 Design Choices in the Semantics of Sequence Diagrams

The semantics given in the UML specification provides the basic ideas of how sequence diagrams should be interpreted. Besides its ambiguity, the specification is also incomplete, i.e., it uses so called semantic variation points, such that part of the semantics is left underspecified, the purpose of this being to allow adapting UML to diverse domains. This flexibility has resulted in a proliferation of alternative formal semantics attached to SDs, and the authors of [141] attempted to provide a synthetic view of the main points of variation and the explicitly or implicitly adopted choices. They selected 13 representative approaches which contain both pioneering works influencing most of the later work, as well as less popular ones which concentrate on some very specific issues. They categorized the semantic choices based on the most common points of concern recurring throughout the surveyed papers.

We briefly outline some of the main semantic choices we adopted and the syntactic constraints they imply, and in that way we position our work in the context of this thorough classification. We provide operational semantics of SDs, implemented in the mCRL2 process algebra, for the purpose of automated model checking of behavioral correctness. Other approaches range from denotational, trace based, Büchi automata, graph-transformation based, and Petri Nets semantics, depending on the purpose and interest for using SDs.

Only a small number of approaches handle *Gates* and *InteractionUse* explicitly, or consider a system design consisting of multiple (related) SDs. Our proposed approach takes both of these aspects into consideration. Furthermore, a non-negligible subset of the proposed semantics handle only synchronous messages, while we take into account asynchronous communication, where the sender of the message can proceed immediately without waiting for a return value.

As far as the trace representation goes, the commonly taken approach is using $m1\_send.m1\_receive.m2\_send.m2\_receive$ or $!m1.?m1.!m2.?m2$, to denote a sequence of events for the sending and receiving of messages. However, as already mentioned in the previous section, in concurrent settings this formal semantics must be more explicit, for cases where several lifelines in an SD are sending the same message to a target one. It should therefore be specified which lifeline (object) sends or receives

the message. Only four of the surveyed approaches [102; 119; 89; 174] represent events of a trace with tuples containing such information. We encode the class, object, and OS process instance to which the exhibited method behavior belongs at runtime, including the data parameters passed along in the message call/reply.

Combined fragments provide a lot of flexibility for modeling programming language-like constructs, but at the same time complicate the understanding of their intended semantics. The semantics of the *loop* fragment is ambiguous with respect to the termination condition, when the guard is expressed as a combination of iteration bounds and a Boolean guard. This ambiguity is a result of two contradictory statements in the UML specification, making it unclear what happens if the condition evaluates to false before the minimum number of iterations have executed. The semantics does not specify at which moment the boolean condition evaluation starts to matter for the loop termination. We treat only the cases where either the iteration bound or the Boolean guard is specified, but not both. According to the UML specification, if several operands within an *alt* fragment evaluate to true, one of them is chosen non-deterministically. Some approaches try to reduce this non-determinism, by evaluating the operands from top to bottom and choosing the first operand evaluating to true. We leave the flexibility of non-deterministic choices in case of multiple true guards. It is also not well defined which lifeline should evaluate the guard and make the choice between the operands of this fragment. Some approaches treat the entering and exiting of fragments as explicit events, and as a consequence all involved lifelines synchronize before entering a choice (making the choice a global one). In contrast, in our methodology the guard is evaluated locally by the lifeline where the first message within the operand is communicated, but the semantics ensures that all involved lifelines execute the same operand. This choice is in line with the STAIRS [102] and the Kster-Filipe [122] surveyed approaches.

With respect to the manner of processing SDs to obtain formal semantics, there seem to be two prevailing approaches. The first category analyze the diagram as a whole, and label the graphical locations of all relevant events, before computing their relative ordering constraints. The second category parse the diagram and decompose it. This is how our approach works. We construct an abstract syntax tree from all the elements of an SD, and then build the formal model by recursively unfolding the fragments and combining them based on the rules defined for each operator of a combined fragment.

The biggest confusion arises around the usage of the conformance-related operators. The semantics defined in our approach assumes that all behavior is explicitly modeled in the diagrams, so we do not take into account the *consider* and *ignore* fragments (more information about their confusing semantics can be found in the UML infrastructure document [92]). Furthermore, we have not encountered implementation-language counterparts in object-oriented distributed software. We

try to stay pragmatic, and avoid constructs which are scarcely used in practice, while they challenge and complicate the semantics unnecessarily.

The *assert* and *negate* fragments alter the way a trace can be classified as valid or invalid, and should be considered as truth modalities, rather than basic behavioral operators, as we shall explain in Chapter 5, where we treat the specification of invalid behavior. The definition of validity or invalidity of behavior depends on the properties that are checked on the model. As such, these two fragments are not covered in the translation presented in this chapter.

4.3.3 Mapping UML to mCRL2

Figure 4.7 gives an overview of our approach and implemented toolset. Both the source (UML) and the target (mCRL2) models adhere to their respective metamodels. Although any UML modeling tool with XMI export/import capabilities can be used, we chose IBM's Rational environment because of the excellent support for SDs and consistency preservation across multiple views of the same model. For parsing and manipulation of the XMI representations we use Eclipse's MDT-UML2 plugin, which implements the UML 2.x metamodel, and provides a Java application programming interface (API) for accessing the "metaobjects" of any imported UML diagrams, and applying further transformations to their in-memory representation. The transformation rules define how to generate a model that conforms to a particular metamodel, from a model that conforms to another metamodel.

To achieve the basic idea of mapping each method along a lifeline into an mCRL2 process description, we process the visually-ordered events along every lifeline individually, thus decomposing the lifeline into individual *ExecutionSpecifications*. We take into account both synchronous and asynchronous messages, so there are es-

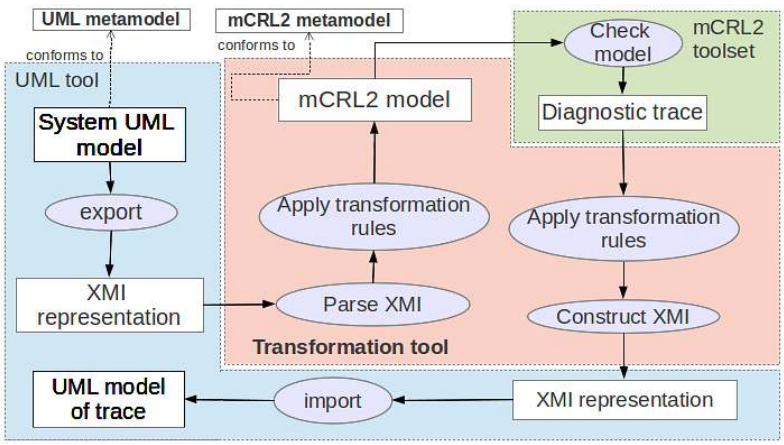


Figure 4.7: Automated verification of UML models

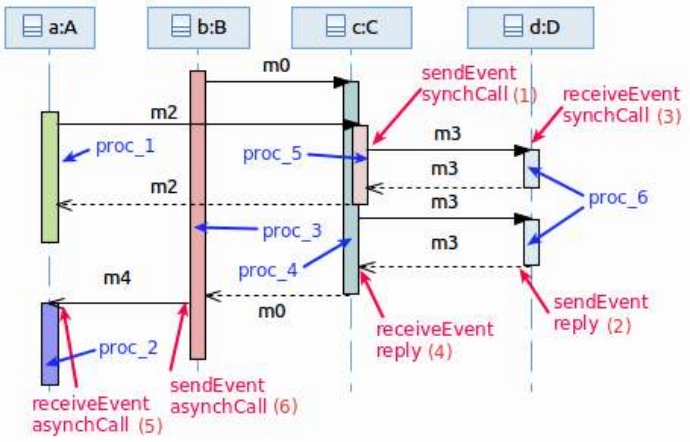


Figure 4.8: Identifying event types along lifelines: m0–m4 are messages invoking particular method behaviors (ExecutionSpecifications), each message being identified by two events; proc_1–proc_6 are identifiers of the corresponding mCRL2 process descriptions.

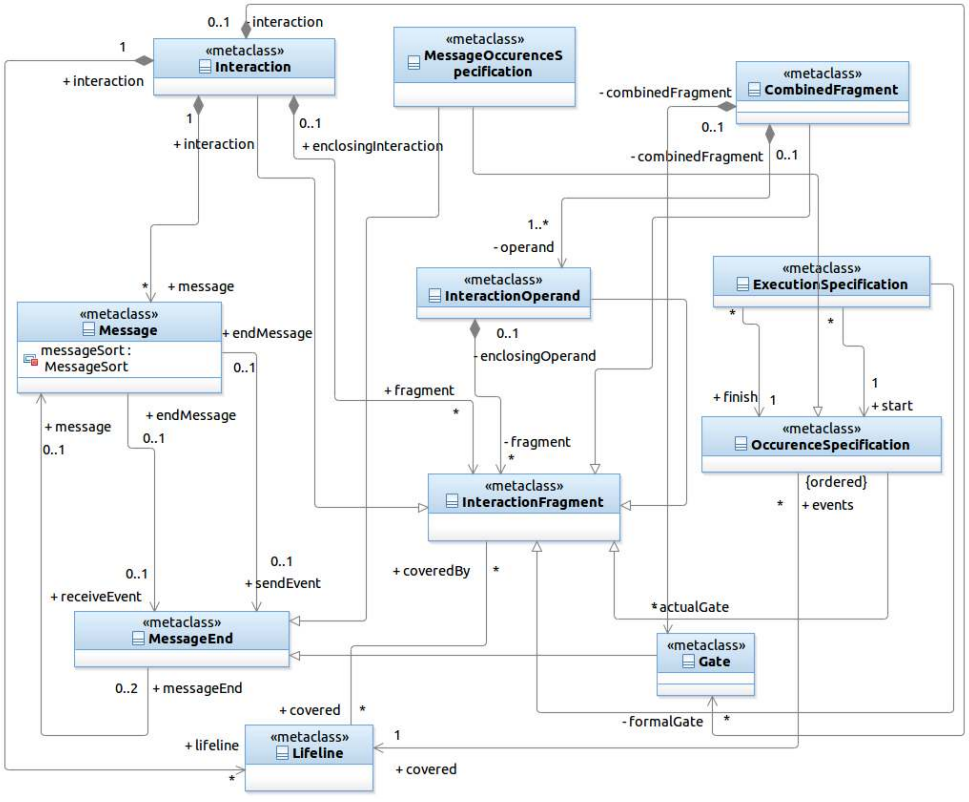


Figure 4.9: Selected elements of the Interactions metamodel

sentially 6 different types of message events (shown in Figure 4.8) that we consider: (1) *SendEvent_synchCall*; (2) *SendEvent_reply*; (3) *ReceiveEvent_synchCall*; (4) *ReceiveEvent_reply*; (5) *ReceiveEvent_asynchCall*; and (6) *SendEvent_asynchCall*. In UML metamodel terms, each of these events corresponds to *MessageOccurrenceSpecifications*, and refers to one of the two ends of each *Message*. To understand the relations between these model elements, we use a simplified class diagram of the Interactions metamodel in Figure 4.9. An *Interaction* (a sequence diagram) essentially encloses *Messages*, *Lifelines* and an ordered list of *InteractionFragments*. Each *Message* is accompanied by a pair of *MessageOccurrenceSpecifications*, and has a reference to the lifelines at which the message is sent and received. Both the *MessageOccurrenceSpecification* and *CombinedFragment* elements are specializations of *InteractionFragment*. We exploit these relationships in our algorithm for matching and transforming into an mCRL2 model. The main algorithm steps are given in continuation.

All fragments of an *Interaction* are processed sequentially, and depending on their type, different mapping rules are applied. In case of *MessageOccurrenceSpecifications*, each event type is treated separately. In case of a *CombinedFragment*, each *InteractionOperand*'s nested fragments are treated by applying the same procedure recursively. Two in-memory stacks are kept for the currently "ready" and the "busy" methods on each lifeline, for cases of overlapping invocations. Busy methods are waiting (blocked) for a reply from another method execution that they have invoked, while ready processes are active, but not blocked. At the beginning, the message, arguments, class, and object corresponding to the handling event are retrieved from the current *fragment* object:

```
1: procedure processFragments(fragments ← interaction.getFragments())
2:     for each fragment in fragments do
3:         if fragment.type = MessageOccurrenceSpecification then
4:             message ← fragment.getMessage()
5:             arguments ← message.getArguments()
6:             event ← fragment.getEvent()
7:             objName,className ← fragment.getCovered.getObjectAndClass()
8:             theReadyStack ← readyProcessesPerLifeline.get(objName)
9:             theBusyStack ← busyProcessesPerLifeline.get(objName)
```

The below pseudocode handles the case of *SendEvent_synchCall* observed on a lifeline. For invocation to be possible, the object representing that lifeline must already be active in some method, at the same time **not** being blocked and awaiting for a return from a method call. We obtain that "ready" method (or mCRL2 *process*) from a stack, on line 12. In addition, this is the only valid UML case where it is possible for a *SendEvent_synchCall* to be the first event inside a *CombinedFragment*. The different fragment types are handled by associating a corresponding mCRL2 structure in the mCRL2 process (lines 16-30). The details of how each type of fragment is

mapped to an mCRL2 structure will be explained after the algorithm walk-through, where also invocations (line 32) added to each process will be discussed.

```
10:          switch event do
11:              case SendEvent_synchCall :                          ▷ Case (1)
12:                  mcrl2Process ← theReadyStack.pop()
13:                  if insideInteractionOperand & firstEvent then
14:                      operator ← getCombinedFragmentOperand()
15:                      guard ← getCombinedFragmentOperandGuard()
16:                      if operator = "alt" then
17:                          mcrl2Process.addAltFragment(guard)
18:                      else if operator = "opt" then
19:                          mcrl2Process.addOptFragment(guard)
20:                      else if operator = "break" then
21:                          mcrl2Process.addBreakFragment(guard)
22:                      else if operator = "par" then
23:                          mcrl2Process.addParFragment(guard)
24:                      else if operator = "loop" then
25:                          loopProcess ← newLoopProcess()
26:                          mcrl2Process.addCallToLoopProcess(loopProcess)
27:                          theReadyStack.push(mcrl2Process)
28:                          loopProcess.addCondition(guard)
29:                          mcrl2Process ← loopProcess
30:                      end if
31:                  end if
32:                  mcrl2Process.addInvocation(
    "synch_call_send(id, className, objName, message, args)")
33:                  theBusyStack.push(mcrl2Process)
```

Once a method sends a reply (*SendEvent_reply*), that mCRL2 process description is finished. The process is removed from the appropriate "ready" stack, as it no longer exhibits behavior after this point:

```
34:              case SendEvent_reply :                              ▷ Case (2)
35:                  mcrl2Process ← theReadyStack.pop()
36:                  mcrl2Process.setProcessed()
37:                  mcrl2Process.addInvocation("
    synch_reply_send(id, className, objName, message, args)")
```

The event of receiving a synchronous call on a lifeline indicates a method invocation. Unless the mCRL2 process corresponding to this method has already been fully constructed ("processed"), a new one is created. Finally the process is pushed to the "ready" stack, since this method is now active (executing), but not blocked:

38: **case** *ReceiveEvent_synchCall* : ▷ Case (3)
39: *findProcess* ← *findProcess*(*className*, *message*)
40: **if** *findProcess* = *null* **then**
41: *findProcess* ← *newProcess*(*className*, *message*)
42: **end if**
43: **if** ¬*findProcess.isProcessed* **then**
44: *findProcess.addInvocation*(
"*synch_call_receive*(*id*, *className*, *objName*, *message*, *args*)")
45: **end if**
46: *theReadyStack.push*(*findProcess*)

To get an intuition on how the algorithm proceeds further, we treat Cases 4, 5, and 6 along with a graphical notation (Figure 4.10), rather than an algorithmic exposition. Upon reception of a reply from a method call (Case (4)), the one initiating it is no longer blocked, so the corresponding mCRL2 process is removed from the "busy" stack and added to the "ready" one. Handling of Case (5) is analogous to Case (3), except that a different kind of invocation is added to the mCRL2 process. Finally, handling Case (6) is also analogous to Case (1), with the important difference being that the active method invoking this asynchronous call on another object will not be blocked after the call. This is why the process is not removed from the "ready" stack nor pushed to the busy one.

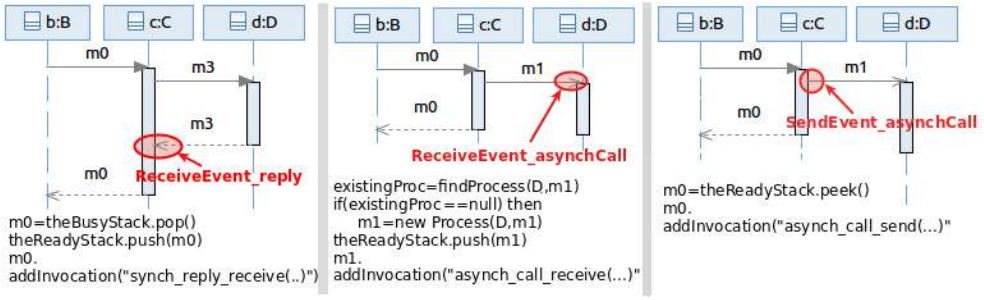


Figure 4.10: Handling Case 4(left), Case 5(middle) and Case 6(right)

All operands that belong to a *CombinedFragment* are processed in turn, recursively handling all fragments (possibly also nested *CombinedFragments*) contained in them, by calling *processFragments* again:

47: **else if** *fragment.type* = *CombinedFragment* **then**
48: **processCombinedFragment**(*fragment*)
49: **end if**
50: **end for**
51: **end procedure**

```
1: procedure processCombinedFragment(fragment)
2:     operands ← fragment.getOperands()
3:     for each operand in operands do
4:         processFragments(operand.getFragments())          ▷ handle recursively
5:     end for
6: end procedure
```

This concludes the basic algorithm for transformation of SDs of arbitrary complexity into mCRL2 process descriptions. When the algorithm is applied to the example in Figure 4.8 it results in 6 different process definitions. Although omitted from the transformation rules described so far, *Gates* are merely specializations of *MessageEnds* (Figure 4.9) and are treated in a completely analogous manner. Care is taken to pass the actual parameters' values to the referenced SD.

Profiles are a standard way to extend UML for expressing concepts not included in the official metamodel, or for refining semantic variation points. In short, UML profiles consist of *stereotypes* that can be applied to any UML model elements, like classes, associations, or messages. The control flow is influenced by the data arguments carried in the communication exchanges between objects, and they are typically modeled as entity elements. These are special classes with an *«entity»* stereotype applied to them. This stereotype is commonly used to model patterns like the Model View Controller one [197], but can be applied in general to model any long-lived, passive element that is responsible for holding some meaningful chunk of data, exposed and modified through getter/setter methods. We identify such entities using this stereotype mechanism, and automatically construct the recursive "memory" processes within the mCRL2 specification, which carry all data values (class attributes) as parameters, in a manner that was outlined in Section 3.4.2. Most of the primitive data types used have a direct mapping into mCRL2 types. Lists of data elements can be constructed where the class attributes have a multiplicity larger than 1. Strings are handled using mCRL2's abstract data type capabilities. This holds also for all class and object names encountered in the transformation, as we will show through an example. Due to their simplicity, we will not explain the transformation rules for the activity diagrams constructs used in the transformation; rather, we will demonstrate them on an example.

The different invocation types added to the mCRL2 processes in the course of the transformation (e.g., line 32, 37, 44) are mCRL2 actions. They carry all parameters necessary for exchange of data between processes, and are used for synchronizing the processes on the corresponding send/receive events. The actions *synch_call_send* and *synch_call_receive* represent two ends of a synchronous message exchanged between two processes. Similarly, *synch_reply_send* and *synch_reply_receive* correspond to a reply message, while *asynch_call_send* and *asynch_call_receive* represent an asyn-

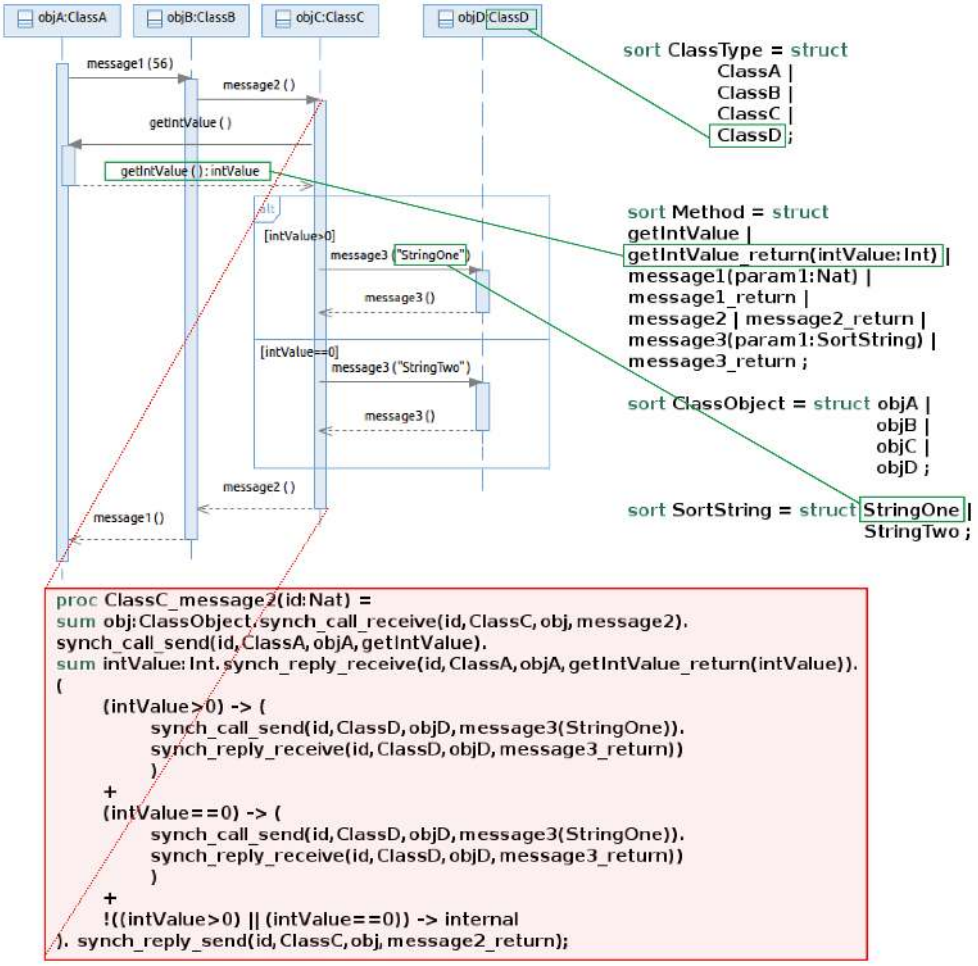


Figure 4.11: Application of the SD transformation rules

chronous call. By applying the mCRL2 communication (Γ) and allow (∇) operator in the following manner:

$$\nabla_{\{synch\_call,\ synch\_reply,\ asynch\_call\}}\Gamma_{\{synch\_call\_send|synch\_call\_receive \rightarrow synch\_call,\ synch\_reply\_send|synch\_reply\_receive \rightarrow synch\_reply,\ asynch\_call\_send|asynch\_call\_receive \rightarrow asynch\_call\}}$$

communication is enforced between processes as a result of the multi-action synchronization at the corresponding events.

Figure 4.11 demonstrates the transformation rules on a simple example. Classes, objects, method names and string values are represented by enumerated data types (**struct**). Replies are distinguished from their respective method calls, as they may also carry parameters. The mCRL2 summation (**sum)** operator is used for bind-

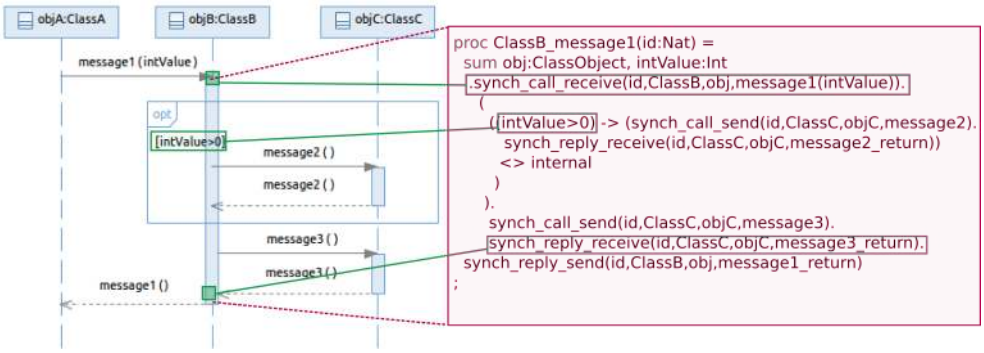


Figure 4.12: Translation of the opt combined fragment

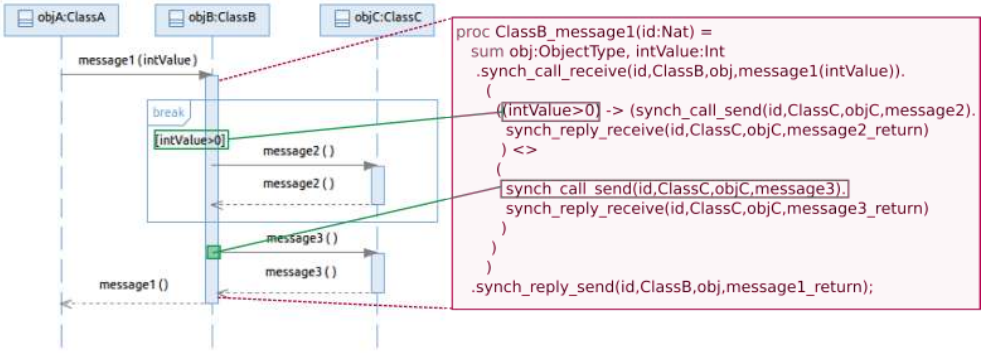


Figure 4.13: Translation of the break combined fragment

ing parameter identifiers to actual values when two processes communicate. This example illustrates how the *alt* fragment is translated into mCRL2 guarded (conditional) non-deterministic choices. To avoid deadlocks, and permit the process to continue in case none of the guards evaluates to true, the **internal** unobservable (τ) action is added as a last "artificial" choice, with the negation of the disjunction of all other guards as a condition. Note that the introduction of this internal step is deliberate, in order to match the mCRL2 *if-then-else* operator with UML's *alt* fragment semantics, and as such, should not mask proper deadlocks. The mCRL2 translation matches the UML semantics, in the sense that, if multiple guards evaluate to true, one will be chosen non-deterministically, whereas, if none evaluates to true, the internal step will be taken, followed by the rest of the scenario in the enclosing fragment. If there is an operand guarded with an explicit "else" condition, that interaction operand's behavior is modeled, instead of the internal step.

The translation of the *opt* and *break* fragments uses the conditional operator in a very similar manner. If the guard of an *opt* combined fragment evaluates to false, the **internal** action is executed, followed by the rest of the scenario (Figure 4.12). If the guard of a *break* fragment evaluates to true, the actions inside the fragment are

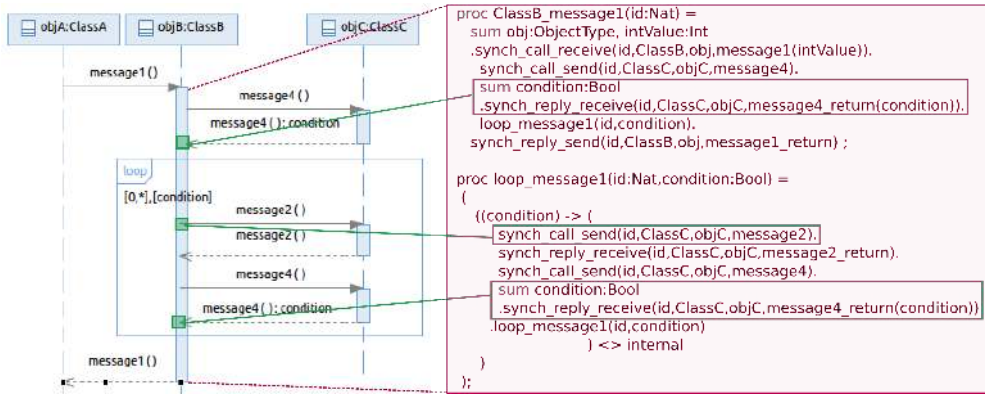


Figure 4.14: Translation of the loop combined fragment

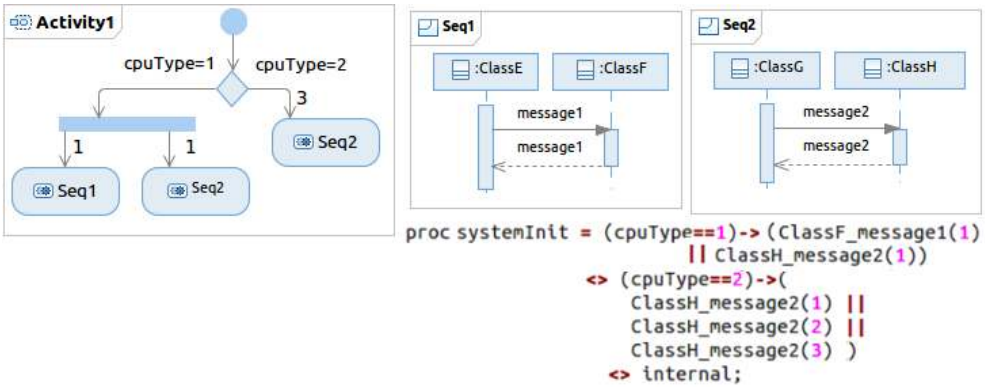


Figure 4.15: Application of AD transformation rules for system-level concurrency setup

executed, and the execution in the continuation of the enclosing fragment is ignored (Figure 4.13). In a *par* fragment, all communications inside each operand are set to run in parallel with the "||" mCRL2 operator. The *loop* fragment has a special treatment. It is translated into a recursive process referenced by the mCRL2 process representing the method active at the moment of entering the loop (Figure 4.14).

Finally, Figure 4.15 shows how simple system-level concurrency can be expressed in an activity diagram, and how it is translated into an mCRL2 specification. ADs have a similar transformation process, namely: *decision nodes* are translated into mCRL2 conditions, *fork/join nodes* are translated using the mCRL2 parallel operator, while *sequential flow* is translated using the mCRL2 sequential composition operator. Weights on the *control flow edges* indicate how many instances of each sequence diagram should be created. In the given example, depending on the condition, three concurrent OS process instances are started with the *Seq2* SD, or alternatively one instance of *Seq1* and *Seq2* in parallel. The *id:Nat* parameter that each mCRL2 process carries is used to bind it to an OS process in the system setup. It should be

noted that the current mCRL2 implementation does not allow the use of "||" unless it is at the topmost level, gluing components together (in the *init* section). This current technical limitation of the tool can be bypassed by adjusting the transformation with explicit *fork_send* (and *join_send*, if the AD uses *Join* nodes) actions in the *systemInit* process, which synchronize with *fork_receive* and *join_receive* action counterparts in the parallel processes, while shifting the parallel processes composition to the topmost *init* section of the mCRL2 model specification.

Validating the Transformation

The current UML specification uses a combination of semi-formal diagrams, a constraints language, and natural language descriptions. While there has been a substantial effort [37; 127; 64] on formalization, there is still no official mathematically-formalized semantics definition. In this context, formal correctness proofs to support the validity of the transformation approach are not possible. To assess the correctness of our proposed semantics, we used the mCRL2 simulation and visualization tools on simple diagrams in isolation, in order to get confidence before applying the transformation methodology on more complex ones gradually. This informal validation was based on the compositional nature of UML and of mCRL2; we closely examined how the basic UML constructs' behavior is reflected in the formal model, with a special focus on combined fragments. Furthermore, to facilitate the transformation, we have constructed a UML metamodel [162] of the mCRL2 language syntax. The fact that we were able to reliably explain complicated bugs observed in practice, provided additional confidence.

4.4 Case Study: DIRAC's Executor Framework

As already explained in Chapter 2, the DIRAC framework was built around three main component types: agents, services, and databases. Agents continuously loop and request certain database entities' states, and react to them by triggering some actions according to their implemented logic. If there are no actions to be executed, agents typically sleep for some configurable amount of time before running the next loop. While this mechanism is effective in most situations, where agents are deployed remote to the centralized DIRAC services and databases, there are certain drawbacks for agents which are running centrally. In particular, the chain of optimizer agents, responsible for sanity checks of jobs before they run on the grid, suffered from poor responsiveness under heavy load, due to this polling mechanism. Namely, the response time of each optimizer agent depends on the sleeping time between each loop cycle. On the other hand, reducing the sleeping time increases the load on the databases whose entities are being requested.

In order to improve the response time without adding extra load to central services and databases, new components called *Executors* had been developed. They register to a central *ExecutorDispatcher* component, and wait for new tasks to be dispatched to them, instead of looping and polling continuously. Again, each Executor is responsible for a different step in the handling of tasks (such as resolving the input data for a job). The ExecutorDispatcher takes care of persisting the state of the tasks and distributing them amongst all the Executors, based on the requirements of each task. It maintains a queue of tasks waiting to be processed, and other internal data structures to keep track of the distribution of tasks among the Executors. During testing, certain problems have manifested in this new framework: occasionally, tasks submitted in the system would not get dispatched, despite the fact that their responsible Executors were idle at the moment. The root cause of this problem could not be identified by testing with different workload scenarios, nor by analysis of the generated logs.

We used our toolset to generate an mCRL2 model, based on UML models that we reverse-engineered from the Executor Framework implementation. Formulas expressed in μ-calculus are again necessary to reason about the correctness of the generated model behavior. Its *action-based* expressions are a good match for our transformation methodology: actions correspond to message exchanges, which already contain the information on the sender/receiver class and object of a particular message. As will be demonstrated, we utilize this for expressing counter-example traces as sequence diagrams. The above-mentioned Executors problem can be formulated as the following safety property in μ-calculus:

$$[true^*.\ synch\_call(1, ExecutorQueues, \_queues, pushTask(JobPath, taskId, false))$$
$$.\ true^*.\ (!synch\_call(1, ExecutorQueues, \_queues\ , popTask([JobPath])))^*$$
$$.\ synch\_reply(1, ExecutorDispatcher, \_eDispatch, \_sendTaskToExecutor\_return(OK, 0))]\ false$$

meaning that a task pushed in the queue must be processed, i.e., removed from the queue before the ExecutorDispatcher declares that there are no more tasks for processing.

Explicit model checking was not feasible in this case due to the model size (50 concurrent processes), so we resorted to using a manually-constructed standard monitoring process, set to run in parallel with the original model and observe the relevant actions that the system itself takes, firing an *error* action in case the property is violated (see Section 3.5.3 for the basic idea). This technique, combined with a depth-first traversal choice in the mCRL2 tool, effectively discovered a trace violating the property within minutes. The counter-example (Figure 4.16) is a rather long trace (available at [162]), and traverses a large fraction of the SDs (27 in total), but the bug was localized in a single SD, covering the behavior of the ExecutorDispatcher component. It is worth mentioning that a breadth-bounded traversal [182] (also known as a highway-search) produced a shorter counter-example trace, however,

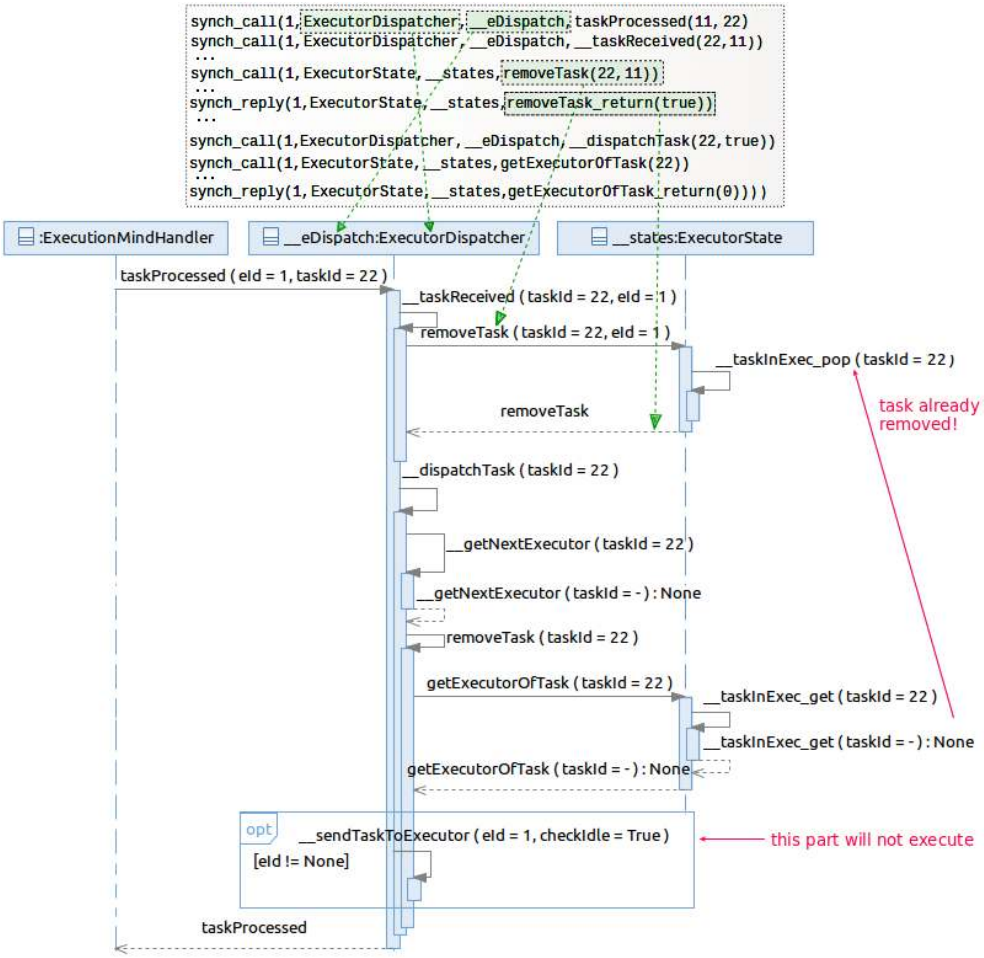


Figure 4.16: SD trace showing a case of no-progress of tasks scheduling

the flaw was still relatively "deep" (144 steps), and the tool took significantly longer (~50 min.) to locate it. We only present the most important part for understanding the cause.

Whenever a task has been processed by some of the Executors, the ExecutorDispatcher is notified (*taskProcessed(eId,taskId)*). To further dispatch the task to another Executor, this task is removed from the ExecutorDispatcher's internal memory of processing tasks, followed by retrieval of the next responsible Executor. In case it was actually processed by the last Executor in the chain, the dispatcher attempts to retrieve its last Executor (*getExecutorOfTask(taskId)*), so that more tasks can be dispatched by this (now free) Executor. However, this information is already removed, as can be seen from the figure. As a result, the *opt* fragment (shown only for clarity, not generated by the toolset) will not be executed, and no further tasks waiting for

this particular Executor will be dispatched from the queue. That would leave the Executor idle for an unpredictable amount of time, until another event (arrival of a new task to the ExecutorDispatcher) triggers filling it again. The bug was reported and fixed. The figure also demonstrates how the trace is translated back in the UML domain, to facilitate debugging the actual problem. A trace is essentially a subset of the behavioral model, and converting it to a sequence diagram can also be seen as a model transformation. The process is relatively simple, since each action of the trace already contains the information necessary to rebuild the message exchange in the sequence diagram. Counter-example traces tend to be lengthy sometimes, making them less suitable for visual representation. A simple post-processing algorithm was implemented, to reduce them before conversion to the UML domain. We keep a configurable number n of leading message exchanges before, and a number m of trailing ones after those matching the actions that are part of the property specification, discarding the rest. This simple approach does not always guarantee that all the crucial trace information for understanding the root cause is kept, but a more involved heuristic was out of the scope for this work.

4.5 Discussion and Evaluation of the Approach

While the case study in the previous section provides evidence that the transformation is effective, in this section we reflect on the efficiency of the methodology, and our experience on the consequences of using it. In contrast to the manual formal modeling we did in Chapter 3, in the process of creating UML models we had much smoother communication and feedback from the DIRAC engineer responsible for developing the Executors Framework. The sequence diagrams were intuitive enough for both parties to convey and mutually agree on the intended behavior in a more straightforward visual manner. Besides being used to obtain a formal specification for model checking purposes, these graphical notations remain as useful documentation artifact for other engineers who wish to understand the dynamics of the module without having to analyze the implementation.

Furthermore, engineers did not have to inspect the produced formal model, in order to interpret and understand the counter-example. All the necessary information (classes, objects, messages, as well as the data they carry) is available in the SD representation of the counter-example, making it easy to trace the problem at the level of the Python code. The flexibility of the UML modeling environment made it easier and more efficient to make modifications and re-check the generated formal model. A current technical limitation of the transformation toolset is the necessity to verify the output model outside of the Eclipse environment. Ideally, mCRL2 should be interfaced with Eclipse, in order to run the model checking transparently within the UML modeling environment.

4.6 Conclusions

In this chapter we tackle **Research Questions 3.1** and (partly) **3.2**. Starting with the assumption that UML is used in common software engineering practice, we have presented an automated transformation methodology for verification of UML models, based on UML sequence and activity diagrams, preserving the object-oriented view of the system in the transformation to a formal mCRL2 model. In our experience, sequence diagrams are the most commonly used graphical notations for specifying the dynamics of a system in terms of interactions between objects over time. Both the input and the output (potential counter-examples) of our implemented toolchain are expressed in UML.

Our proposed approach is based on the UML 2.x semantics, and we outline the semantic choices we made with respect to some ambiguous points in the semantics defined by the OMG. This way we position our work in the context of a well-known classification, and we compare our approach with the existing ones. We cover more of the UML 2.x sequence diagrams constructs than most of the surveyed papers in the classification, in particular the combined fragments, as well as asynchronous communication. Defining the formal semantics of sequence diagrams merely in terms of pairs of valid and invalid traces (trace semantics), like the UML standard prescribes, would not allow us to faithfully keep information about decision moments (see Section 3.2 for the possible effects of this). With our transformation, branching points are visible in the semantics model.

We have further applied our toolset to a component of the DIRAC grid system, and discovered a logical flaw leading to no-progress. Although providing a formal correctness proof of the transformation is impossible, given that there is no official formal UML semantics on sequence diagrams, the fact that we were able to reliably explain this flaw, provides some confidence in the correctness and usefulness of the methodology. The case study in this chapter strengthens the evidence on the advantages of model checking that were already mentioned in Chapter 3, in answer to **Research Question 2**. While the mCRL2 toolset can automatically discover deadlocks and search for specific events, for the purpose of discovering the particular problem encountered in DIRAC, it was necessary to formulate the application-specific property in μ-calculus. This still requires some degree of maturity with the language, and should be made more accessible to software engineers.

Chapter 5

Assisting Non-Experts in Property Specification

5.1 Introduction

Bridging the gap between industry-adopted methodologies based on UML software designs and the aforementioned tools and languages, in Chapter 4 we devised a methodology for automatically verifying UML models comprising sequence and activity diagrams. Our prototype uses the mCRL2 language and toolset as a black box, without users having to leave the UML domain. While the mCRL2 toolset can automatically discover deadlocks and search for specific events, its model checking facilities require users to formalize their application-specific properties in a data-enriched extension of the modal μ-calculus. A downside is that it is not very accessible and requires a high degree of mathematical maturity. As already simpler languages such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) are not widespread in industry, the μ-calculus is unlikely to be easily embraced by it. In fact, most requirements are written in natural language, and often contain ambiguities which make it difficult even for experienced practitioners to capture them accurately in any temporal logic. There are subtle but crucial details which are often overlooked and need to be carefully considered in order to distill the right formula. In an attempt to ease the use of temporal logic, a pattern-based classification was developed in [67] for capturing requirements and generating input to model checking tools. The authors observed that almost all (> 500) properties they surveyed can be mapped into one of several property patterns. Each pattern is a high-level, formalism-independent abstraction and captures a commonly occurring requirement. Their hierarchical taxonomy is based on the idea that each pattern has a *scope*, which defines the extent of program execution over which the pattern

must hold, and a *behavior*, which describes the intent of the pattern. The pattern system identifies 5 scopes and 11 behavior variations that can be combined to create 55 different property templates. Examples of scopes are: *globally*, and *after* an event or state occurs; examples of behavior classification are: *absence* (an event or state should never occur during an execution) and *response* (an event or state must be followed by another event or state).

Although the patterns website contains a collection of mappings for different target formalisms such as LTL and CTL, in practice practitioners have to fully understand the classification before they can select and apply the appropriate pattern. To mitigate this problem, several conversational tools [51; 120; 142] have been proposed for elucidating properties, based on the patterns. These tools guide users in selecting the appropriate pattern and optionally produce a formula in some target temporal logic. Alternative approaches [15; 128; 179; 129; 121; 20] tackle the property specification problem by proposing new graphical notations for specifying properties. As far as we have been able to trace, all approaches deal with state-based logics. Such logics conceptually do not match the typical event-based UML sequence diagrams, in which events represent method calls or asynchronous communication between distributed components. A notable exception is [136], where an extension of the alternation-free μ-calculus with action-based CTL-like formulas and regular expressions is introduced and implemented within the CADP toolbox [81], allowing the description of safety, liveness, and fairness properties via predefined templates. However, no graphical notations or elicitation tool support is provided in choosing the right template for a particular property.

The contribution of our work in this chapter is a simplification of the process of specifying functional requirements for event-based systems. Like the model, property specifications need to be understandable and accessible to non-experts, and at the same time mathematically precise to enable automated analysis. We introduce PASS, a Property ASSistant tool that facilitates deriving system properties. Based on natural language, PASS guides users through the elicitation process by asking questions, and providing a set of alternative answers to choose from, narrowing down the scope of the questions to those relevant in the context of the previously provided answers in each subsequent step. Our starting point was the pattern system [67] and the μ-calculus mappings [134] provided by the CADP team, which we extended with over 150 property templates. The pattern templates instantiated with PASS have three notations: a natural language summary, a μ-calculus formula and a UML sequence diagram depicting desired behaviors. The natural language summary is still readable for non-experts, while the μ-calculus formula can be given as input to the mCRL2 toolset for model checking. Sequence diagrams provide an intuitive level of abstraction for visualizing requirements, by observing the sequence of method calls that should occur in a particular order at runtime. We utilized

mCRL2's rich data extensions of the μ-calculus to express data-dependent properties. Lastly, for a fragment of our properties we automatically generate monitors, which can be used for property-driven on-the-fly verification using the standard exploration facilities of mCRL2. Our monitors are essentially sequence diagrams, acting as observers of message exchanges.

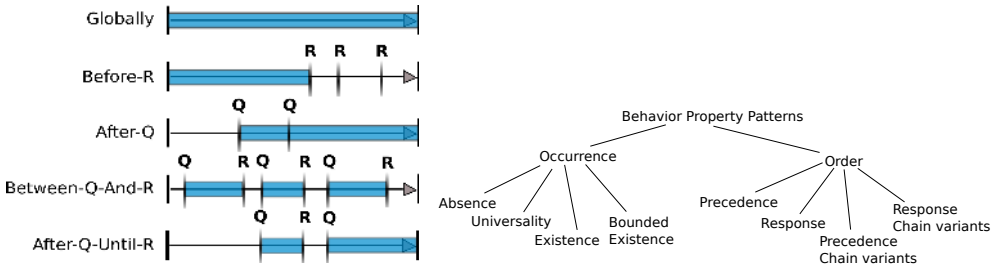
We deliberately chose to develop PASS as an Eclipse plug-in, as our strong motivation was to stay within an existing UML development environment, rather than use an external helper tool for this. This increases the tool's accessibility by allowing software engineers to remain focused in the realm of UML designs. In addition, a tight connection between elements of the design and instances of the property template is kept, such that, if the design is changed, these changes can be easily propagated in the property template placeholders. To this end, we use the standard MDT-UML2 [181] Eclipse modeling API.

We further surveyed 25 published works that use μ-calculus to express system properties from different domains, for two purposes: to reassess the usefulness of the pattern-based classification in event-based systems, with a focus on the μ-calculus as a target formalism, and to indicate how error-prone the manual elicitation of properties can be, even when it has been thoroughly reviewed. This survey strengthens the claim of Dwyer et al. [67] that very few scopes and patterns are sufficient to express the majority of properties. From a total of 178 properties encountered in the publications, around 70% could be fitted within the standard Property Specification Patterns (PSP) classification. If our extensions are taken into account, the coverage is improved by 10%. The results also illustrate that subtle mistakes can creep into complex formulas and potentially hide problems in the studied system model. We encountered 6 publications with wrong (or suboptimal) property formalizations, where "wrong" means "not conveying the intention of the written natural language requirement".

The remainder of this chapter is organized as follows. Section 5.2 briefly introduces the basic Property Specification Patterns classification. In Section 5.3 we discuss related approaches, and outline their advantages and shortcomings. We describe our approach in Section 5.4, with focus on the pattern extensions. In Section 5.5 we apply PASS on case studies of two DIRAC subsystems, namely the SMS and WMS introduced in Chapter 2. In Section 5.6 we report our findings on a large body of published work where properties in μ-calculus have been manually written, and we conclude in Section 5.7.

5.2 Background: Property Specification Patterns

The PSP classification taxonomy [67] was developed as a result of a large survey of formal property specifications, having recognized that they can be organized



(a): Scopes (b): Behaviors

Figure 5.1: Property Specification Patterns classification

into several categories, based on two orthogonal concepts: their *scope* and *behavior*. The patterns are general and can be applied to events or states, depending on the formalism used for specification. In this work we are concerned with event-based systems. The available scope intervals for a property are shown in Figure 5.1a. They can have a beginning, an ending delimiter, or both. In the figure, the horizontal lines represent the model execution timeline during which events occur. The vertical lines labeled with Q and R denote events that mark the beginning and ending of a scope respectively. If a property should hold throughout the entire system execution, then there are no delimiters (*globally*). The *before-R* scope restriction means that the behavior must hold before the first occurrence of R. Similarly, *after-Q* requires that the behavior must hold from the first occurrence of Q onwards. *Between-Q-and-R* is a combination of the former two, i.e., the behavior must hold from the first occurrence of Q until the first subsequent occurrence of R. If the occurrence of R is optional, in the sense that, if R never happens, the behavior must hold until the end of the run, then the appropriate scope is *after-Q-until-R*.

The taxonomy organizes the behavior patterns (Figure 5.1b) in two main categories: *occurrence* and *order*. *Occurrence* patterns are concerned with a single event, and are briefly described as follows. *Absence*: an event must never occur in the system execution; *universality*: an event must always occur in the system execution; *existence*: an event must eventually occur; *bounded existence*: an event can occur at most k times in the system execution. *Order* patterns express properties where the relative ordering of multiple events is important. The two basic ones, *precedence* and *response*, are very common in concurrent systems specifications. The *chain variants* can be considered as generalizations of these, where the precedence/response relationship consists of sequences of more than two events. *Precedence* is used to specify that the occurrence of one event is necessarily preceded by an occurrence of another event. *Response* describes a cause-effect relationship: an occurrence of one event must be followed by an occurrence of another.

The available formalism-specific pattern mappings [67] smoothen the use of temporal logic for novices, but choosing the right template to capture a certain property

is not always straightforward. For example, consider the following requirement:

A component (C2) must eventually be started after the component (C1) it depends on, is started.

It can be instantiated as an *existence* pattern with an *after-Q* scope, or as a *response* pattern with a *globally* scope. Although both patterns describe related behavior, they are not equivalent. In the first case, the property is already satisfied if after the first start of C1, C2 is started once, and never again. On the other hand, the second property is satisfied if and only if every start of C1 is followed by a start of C2. The right choice also depends on the domain knowledge (for example, whether components can be started/stopped multiple times).

5.3 Related Work

PROPEL [51] is a tool that guides users in selecting the appropriate template from the patterns classification. PROPEL adds new patterns covering subtle aspects not addressed by the patterns classification of [67] (such as considering the effect of multiple occurrences of a cause in a response pattern); at the same time it omits other important ones from the standard classification, such as the *universality*, *bounded existence* and the *chain* patterns. The resulting templates are represented using "disciplined natural language" and visual finite state automata, rather than temporal logic expressions. This limits the possibilities of automated verification. A strong point of PROPEL is the interactive question tree representation that helps the user clarify the intention of the property. In particular, the hierarchical questionnare structure supports isolation of concerns, and covers refinements of the original PSP classification, many of which we shall implement in the context of μ-calculus and UML models, with some modifications and additions. Similar to PROPEL, the tools SPIDER [120] and Prospec [142] extend the original patterns but add compositionality. SPIDER is no longer maintained nor available; the latest version of Prospec that we found and tested (Figure 5.2 left) produces formulas in Future Interval Logic (FIL), not LTL as stated in [142]. While FIL is similar to LTL, it is not as widely used in formal verification tools. Furthermore, some of the LTL temporal operators (until **(U)**, next **(X)**) are not available in FIL, limiting its expressiveness. On the other hand, FIL supports search patterns and interval specifications, and there is a translation [63] from a subset of FIL to LTL.

Approaches that use a graphical notation for specifying properties come closest to the realm of modeling the system behavior. In [128], formulas are represented as acyclic graphs of states and temporal operators as nodes. Technically, the underlying LTL formalism is hidden from the user, but the notation still closely resembles the formalism. As such, it is not very accessible. Another tool, called the TimeLine Editor [179] permits formalizing specific requirements using timeline diagrams. For

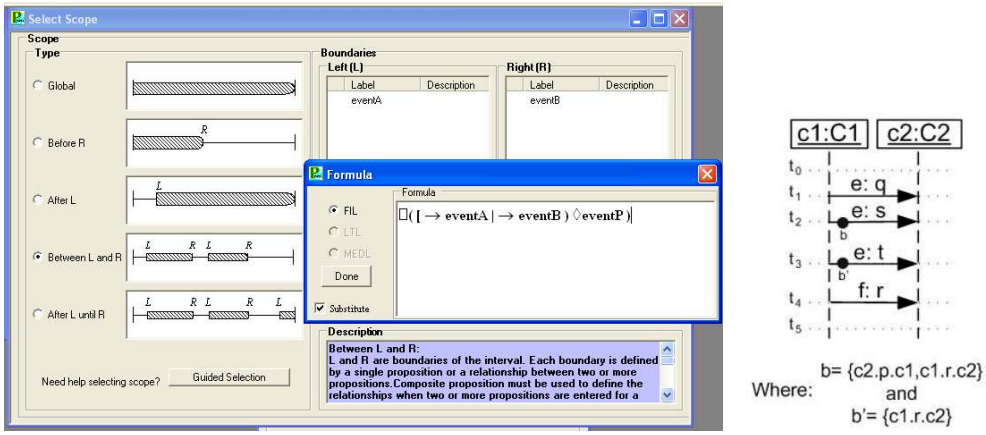


Figure 5.2: Left: Prospec tool; right: CHARMY PSC graphical notation

instance, *response* formulas are depicted in timeline diagrams by specifying temporal relations among events and constraints. These diagrams are then automatically converted into Büchi automata [82], amenable to model checking with SPIN. Unfortunately the TimeLine Editor is no longer available. The CHARMY approach [15] presents a scenario-based visual language called Property Sequence Charts (PSC). Properties in this language are relations on a set of exchanged system messages. The language borrows concepts from UML 2.x sequence diagrams and the tool uses SPIN as a backend for model checking the generated Büchi automata. The PSC notation uses textual restrictions for past and future events, placed as circles directly on message arrows (Figure 5.2 right). A drawback of PSC is that it does not support asynchronous communication, which is omnipresent in concurrent systems. Furthermore, CHARMY is a standalone framework for architectural descriptions, not inter-operable with UML tools. As such, its use in industrial contexts is limited. Another interesting early approach based on the idea of property patterns is described in [114], for the domain of business process modeling. Studying a large number of business cases and identifying the most frequently asked questions has led to 4 categories of patterns. Once instantiated with a graphical tool support, each pattern is translated to LTL and SPIN is used for model checking a business process model representation in PROMELA (the language used by SPIN). Having many similarities, these patterns can in fact be seen as a subset of the PSP system [67].

Among the UML-based tools are HUGO/RT [119] and vUML [129]. HUGO/RT is a tool for model checking UML 2.x interactions against a model composed of message-exchanging state machines. The interactions represent the desired properties, and are translated together with the system model into Büchi automata for model checking with SPIN. The version we tested supports no asynchronous messages nor combined fragments. vUML [129] is, like HUGO/RT, essentially a tool

for automating verifications of UML state machines. Properties must be specified in terms of undesired scenarios. The verification is based on the ability to reach error states. This is inconvenient, as users must specify these manually. In addition, not all properties can be captured in this way. Live Sequence Charts (LSC) are also used [59; 121; 20] as a graphical formalism for expressing behavioral properties. They can distinguish between possible (cold) and mandatory (hot) behaviors. For both, Büchi automata and LTL formulas are generated automatically from the diagrams. UML 2.x sequence diagrams borrow many concepts from LSC, by introducing the *assert* and *negate* fragments for capturing mandatory and forbidden behavior. However, LSCs lack many UML features. Inspired by Message Sequence Charts, in [14] a visual language called Event Sequence Charts with Quantitative Constraints (ESC-QC) is presented for specification of requirements in the domain of embedded systems. ESC-QC specifications can be translated into Simulink/Stateflow monitors and used for run-time verification of safety properties.

Resembling an OO programming language, UML's declarative Object Constraint Language (OCL) is also considered for formal specification purposes. Its original purpose is to specify invariants on classes, and pre- and post conditions on operations. However, constraints quickly become quite dense and cryptic, and editing them manually is error-prone. Another problem is the extent to which designers are familiar with this language. Finally, OCL is by itself incapable of reasoning about temporal behavior. There are several temporal extensions of OCL. In [202] temporal modifiers *@pre* and *@next* are introduced for specifying past and future state-oriented constraints. In [75], a real-time constraint extension of OCL is proposed for models described by UML state machines; it claims to be able to describe all the existing patterns in these OCL expressions. To simplify constraint definitions with OCL, [7] proposes to use specification patterns for which OCL constraints can be generated automatically. The behavioral specification of software components refers to interface specifications, which are not really dynamic views. In effect, this work does not really introduce means to specify temporal properties. An extension of OCL with μ-calculus is proposed in [32], with the aim to combine the temporal features of μ-calculus with the static expressiveness of OCL. The resulting logic, observational μ-calculus, is rather expressive. Nevertheless, as the authors remark, it is unrealistic to expect that UML designers would be interested in becoming proficient in the logic, so they suggest using templates with a designer-friendly syntax, which can then be translated into formal expressions. They demonstrate the idea on one such *after/eventually* template, which actually corresponds to the *response* pattern of the PSP classification. This is an interesting work in the context of a possible/future UML verification tool. However, the task of actually verifying a UML model with such an extended μ-calculus logic, is not tackled.

Finally, in the area of constructing specifications from natural language require-

ments, a recent ambitious work presented in [76] attempts to combine the advantages of model- and natural language-based requirements engineering. It proposes a methodology for bidirectional translation between model-based requirements and controlled natural language (CNL) expressions with a restricted vocabulary. In particular, requirement patterns expressed in CNL can be transformed into SysML internal block diagrams and sequence diagrams, and vice versa. Although useful for property specification, this approach does not yet provide artifacts amenable for automatic analysis and formal verification. A similar, very early work in the area of Natural Language Processing as a means to assist in the formalization of behavioral requirements is [72]. A prototype assistant tool NL2ACTL is presented, which aims to recognize informal behavioral requirements and translate them into the branching time temporal logic ACTL [60] (an action-based CTL flavor). Their grammar can recognize and deal with certain ambiguities, by interacting with the user and asking for the missing information, before generating the formula.

5.4 PASS: An Implementation and Extension of PSP

It has already been recognized that the process of "patterning" the natural language requirements is still challenging. Case studies [155; 74] have indicated some difficulties in introducing the PSP system in its current form in the industrial process. There is a need for softening the learning curve, preferably by accompanying the patterns with support material, such as graphical representations and examples. To describe our proposal to a correct and straightforward property elucidation, we outline the motivations behind the choices we made, and how they differ from existing related approaches.

While we follow on the idea of using a guiding questionnaire to incrementally refine various aspects of a requirement, we find the resulting artifacts (LTL formulas or graphical representations of finite state machines) from using the available tools not yet suitable for practical application in our context. For one, the practitioner must manually define the events to be associated with the placeholders when instantiating the template. To avoid potential errors, as well as reduce the effort of specifying, we want to ideally stay in the same IDE used for modeling the system, and only use existing events that represent valid communication between components.

5.4.1 μ-calculus and the PSP system

Most approaches cover the (state-based) LTL mappings and extensions of the pattern system. Event-based temporal logics have not received much attention. Even though the original pattern system does not cover μ-calculus, such mappings for the

regular alternation-free μ-calculus [134] have been developed by the CADP team, but their effectiveness has not been established. These are adequate for action- or event-based systems, making them a good match for visual scenarios, where communication between components is depicted. LTL is interpreted over Kripke structures, where the states are labeled with elementary propositions that hold in each state, while μ-calculus is interpreted over Labeled Transition Systems (LTS), in which the transitions are labeled with actions that cause the state changes. Even though both are complementary representations of the more general finite state automata, conversions between them are not practical, as they usually lead to a significant state space increase. For example, the fact that a lock has been acquired or released can be naturally expressed by actions. Since state-based temporal logics lack this mechanism, an alternative is to introduce a variable to indicate the status of the lock, i.e., expose the state information. For such properties, LTS representations are more intuitive, and easier to query using event-based logics. LTS structures provide the formal operational semantics for behaviors typically described with process algebras, such as mCRL2.

Brief Introduction to μ-calculus

As already stated, the language used by the mCRL2 toolset for model checking specific properties is an extension of the modal μ-calculus. We informally introduced μ-calculus in Chapter 4; for the purpose of property elicitation we now provide a more complete description of its syntax and semantics. This formalism stands out from most modal and temporal logic formalisms with respect to its expressive power. Temporal logics like LTL, CTL and CTL* all have succinct translations [55] into the first-order μ-calculus (propositional μ-calculus extended with data), witnessing its generality. This expressiveness comes at a cost: very complex formulas with no intuitive and apparent interpretation can be coined. The syntax of mCRL2's modal μ-calculus formulas we are concerned with in this chapter is defined by the following grammar[1]:

$$
\begin{array}{ll}
\phi & ::= b \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall d{:}D.\phi \mid \exists d{:}D.\phi \mid \\
 & [R]\phi \mid \langle R \rangle \phi \mid \mu Z.\, \phi(Z) \mid \nu Z.\, \phi(Z) \mid Z \\
R & ::= \alpha \mid R \cdot R \mid R^* \mid R^+ \\
\alpha & ::= b \mid a(d_1, \ldots, d_n) \mid \neg\alpha \mid \alpha \cap \alpha \mid \alpha \cup \alpha \mid \\
 & \bigcap d{:}D.\alpha \mid \bigcup d{:}D.\alpha
\end{array}
$$

Each μ-calculus formula is either valid or invalid at a state of a Labeled Transition System. Action formulas α describe sets of (multi-)actions; these sets are built

[1] Here the operators are shown in rich text format. The formulas in previous chapters were written in plain (ASCII) format, as used by the toolset. For a mapping between these notations, please refer to [84].

up from the empty set of actions (in case the Boolean expression b evaluates to false), the set of all possible actions (in case b evaluates to true), individual actions $a(d_1, \ldots, d_n)$, action complementation $\neg\alpha$, and finite and possibly infinite intersection $\cap$ and union $\cup$ of actions. The action formula $\bigcup d{:}D.\alpha(d)$ represents the union of all sets of actions obtained by replacing every free variable d with concrete values from the domain D. The modal operators take regular expressions R for describing words of actions, built up from individual actions described by an action formula α, word concatenation $R \cdot R$ and (arbitrary) iteration of words R^* (zero or more repetitions of R) and R^+ (one or more repetitions of R). Properties are expressed by state formulas ϕ, which contain Boolean data terms b that evaluate to true or false and which can contain data variables, the standard logical connectives *and* ($\wedge$), *or* ($\vee$), as well as the derived logical *implication* ($\phi \Rightarrow \psi$, a shorthand for $\neg\phi \vee \psi$, where ϕ does not contain a recursion variable Z), the modal operators *must* ($[\_]\_$) and *may* ($\langle\_\rangle\_$), and the least and greatest fixpoint operators μ and ν. In addition to these, mCRL2's extensions add universal and existential quantifiers $\forall$ and $\exists$. A state of an LTS (described by an mCRL2 process) satisfies $\langle R\rangle\phi$ iff from that state, there is at least one transition sequence matching R, leading to a state satisfying ϕ; $[R]\phi$ is satisfied by a state iff all transition sequences matching R starting in that state lead to states satisfying ϕ. For instance, $[\neg(\bigcup n{:}Nat.\ read(n+n))]$ *false* states that a process should not execute any actions other than read actions with even-valued natural numbers. Note that $[a]\phi$ is trivially satisfied in states with no "a"-transitions.

Combining these modalities, the least ($\mu X.\phi(X)$) and greatest ($\nu X.\phi(X)$) fixpoints permit reasoning about finite and infinite runs of a system in a recursion-like manner. For example, we can read $\mu X.\phi \vee \langle\alpha\rangle X$ as: X is the smallest set of states such that a state is in X iff ϕ holds in that state *or* there is an α-successor in X. On the other hand, $\nu X.\phi \wedge [\alpha]X$ is the largest set of states such that a state is in X iff ϕ holds in that state and all of its α-successors are in X, too.

A Labeled Transition System $A = (S, Act, \longrightarrow, s_0, T)$ satisfies a μ-calculus formula ϕ iff its initial state s_0 satisfies ϕ.

Pattern Mappings for μ-calculus

To get an impression of the μ-calculus pattern mappings [134], we illustrate some commonly used templates. Basic safety properties are generally expressed using the box (*must*) modality containing regular formulas. For example, the *precedence* pattern with a *global* scope is characterized in the following way:

$[(\neg S)^*.\ P]$ *false*

To specify that a memory cleanup must occur before the program shutdown, the generic event placeholders S and P should be substituted with the actual events:

$[(\neg$*clean_memory*$)^*.$ *program_shutdown*$]$ *false*

The formula states that the process should never exhibit a behavior in which a *program_shutdown* event is preceded by an arbitrary number of events, none of which matches *clean_memory*. An *after-Q* scope restriction of the same pattern has the following template:

$[(\neg Q)^*.\ Q.\ (\neg S)^*.\ P]\ \mathit{false}$

The regular expression has been modified to consume all events up to the point of the occurrence of Q.

Liveness[2] properties typically make use of the diamond (*may*) modality and fixed point operators. For instance, the *response* pattern with a *global* scope is expressed as follows:

$[\mathit{true}^*.\ P]\ \mu X.\langle \mathit{true}\rangle \mathit{true} \wedge [\neg S]X$

The template can be instantiated to state that after every *send* event, all execution sequences will eventually lead to the *receive* event:

$[\mathit{true}^*.\ \mathit{send}]\ \mu X.\langle \mathit{true}\rangle \mathit{true} \wedge [\neg \mathit{receive}]X$

An *after-Q* scope restriction of the same pattern has the following template:

$[(\neg Q)^*.\ Q.\ \mathit{true}^*.\ P]\ \mu X.\langle \mathit{true}\rangle \mathit{true} \wedge [\neg S]X$

Linear Time vs. Branching Time

Theoreticians as well as practitioners of temporal logic as a language for specifying properties, are mainly divided in two groups: those who advocate linear time, and those who advocate branching time logics. If we consider Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) as the most widely accepted representatives of both categories, the comparison is not a definite one with a clear winner. With respect to their expressive power and model checking complexity, there are sound arguments in favor of both sides [124; 70; 192; 191], and their relationship has been studied extensively [133; 46]. Their differences stem from the way (discrete) time is interpreted along the unfolding of the model state space. LTL formulas express properties that are interpreted along each single execution (or run). In other words, the formula satisfiability is checked along every run without the possibility of switching to another run during the traversal. On the other hand, CTL formulas express properties that are interpreted on computation trees, rather than linear runs. This notion of branching time means that at any given state during the exploration, when facing a branch, all (or only one, CTL has universal and existential quantification operators) different possible futures are unfolded and explored to check satisfiability.

[2] Properties that require that "something good eventually happens". The *existence* and *response* patterns are an example. Contrary to these, safety properties require that "something bad never happens." All non-trivial functional properties can be classified as safety, liveness, or a mix of these two [11].

Table 5.1: After-Q vs. After-Last-Q scope variations for different behavior patterns

Behavior	**After-Q**	**After-Last-Q**
Absence	$[(\neg Q)^*.\ Q.\ true^*.\ P]\ false$	$[true^*.\ Q.\ (\neg Q)^*]$ $((([true^*.\ Q]\ false) \Rightarrow [true^*.\ P]\ false)$
Existence	$[(\neg Q)^*.\ Q]$ $\mu X.(\langle true\rangle true \wedge [\neg P]X)$	$[true^*.\ Q.\ (\neg Q)^*](([true^*.\ Q]\ false)$ $\Rightarrow \mu X.(\langle true\rangle true \wedge [\neg P]X))$
Precedence	$[(\neg Q)^*.\ Q.\ (\neg P)^*.\ S]\ false$	$[true^*.\ Q.\ (\neg Q)^*]$ $((true^*.\ Q]\ false) \Rightarrow [(\neg P)^*.\ S]\ false)$
Response	$[(\neg Q)^*.\ Q.\ true^*.\ P]$ $\mu X.(\langle true\rangle true \wedge [\neg S]X)$	$[true^*.\ Q.\ (\neg Q)^*](([true^*.\ Q]\ false)$ $\Rightarrow [true^*.\ P]\mu X.(\langle true\rangle true \wedge [\neg S]X))$

5.4.2 Pattern Extensions for μ-calculus

The PSP authors recognized that variations of the proposed scopes and behaviors may be necessary in certain cases, and provided basic notes on how to modify the specifications for some of them. To improve expressiveness without users having to manually tailor their formulas, we have incorporated several pattern extensions in PASS, which were also absent from the μ-calculus mappings [134] of the Figure 5.1 PSP classification, provided by CADP. For instance, as [51] points out, by default the scopes are interpreted with respect to the *first* occurrence of the beginning scope delimiter. Subsequent occurrences of that event are ignored until the occurrence of the end-delimiter closes the interval. Consider the following property:

After the last bid, the product is paid by the winner.

Online auction systems typically receive bids from multiple customers, but only the last one matters, before the auction closes. In such case, there should be an option where the beginning of the scope is essentially reset with every new occurrence of the event Q. We provided a scope variation *after-last-Q* and a similar option for the *between-Q-and-R* scope. While it is not expected that non-experts should be willing or able to understand the potentially complex formulas, Table 5.1 illustrates how this scope was adapted for some of the behaviors. Furthermore, the original taxonomy does not clarify how to treat a behavior combined with a *before-R* scope restriction, if the event R does not happen until the end of the system execution. For such a case, the default semantics provided in [134] are that the specified behavior is not required to hold, i.e., the property is trivially (vacuously) satisfied, **unless** the end-event R is seen. A variation (*after-Q-until-R*) was only provided for the *between-Q-and-R* scope. Consider the following property:

Every request will eventually be followed by a response, until the socket is disconnected.

Since there may exist runs in the model in which the socket is never disconnected, the response pattern behavior (*"Every request will eventually be followed by a response"*) must hold until either the socket is disconnected, or until the end of the run if the socket is never disconnected. We kept the default semantics and provided an option to select an *until-R* scope variation, which is semantically equivalent to *after-True-until-R*. The added scope variations are shown in Figure 5.3a.

Given that communication among components proceeds via actions (or events) which can represent synchronous or asynchronous communication, a property specification can be defined over sequences of actions that carry data. Fortunately, the mCRL2 μ-calculus is rich enough to express both state and action formulas, and provides means for quantification over data, which many formalisms lack. With our approach, a practitioner can use a wildcard "*" to express that the property should be evaluated for all values that message parameters can carry. This allows us to use patterns which would otherwise make sense only for state-based formalisms. For example, in its original form, the *universality* behavior pattern describes a portion of the system's execution which contains only states/events that have a desired property. In event-based systems this would directly translate to *"an event must always occur in the system execution"*, which is not a very useful property for a system to exhibit, since it means that there is only one possible event always taking place. Even though this pattern is provided as such in [134], a more practical alternative is checking if an event always carries the same data value, so we refined it in this context. For comparison, the provided pattern template for a *global* scope is:

$[true^*.\neg P]\ false$

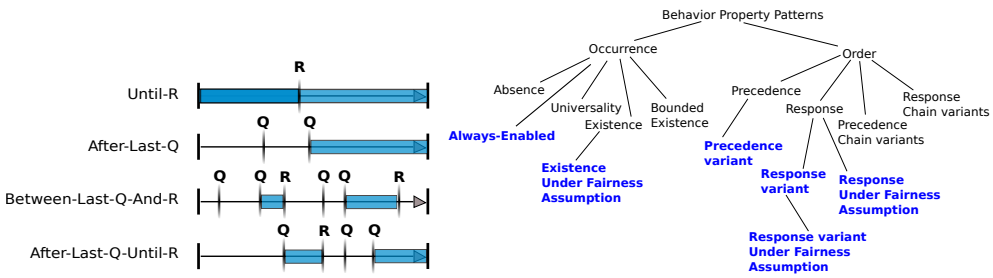
which means that at most event P can happen at any moment, while:

$[true^*]\ \forall d{:}D.(d \neq d1) \Rightarrow\ [P(d)]\ false$

means that the event P can only carry a data value *d1*. Related to this pattern, an interesting and realistic property is to ask if some event is always possible (or enabled), rather than always taken. For example, "*it should always be possible to exit a program*". For this purpose, we have added the *always-enabled* pattern. For example:

$[(\neg initialize)^*.\ initialize.\ (\neg exit)^*]\langle exit\rangle true$

means that at any point after initialization, the *exit* event is enabled. As Leslie Lamport points out in [125], such "possibility" properties are argued as less interesting compared to knowing that the system will never produce a wrong answer (*safety*) and that it eventually will produce an answer (*liveness*). However, if in a formal model of a nuclear power plant it were impossible to require a shutdown from any reachable state, then the model would be wrong. Even if it seems unnecessary to check that a user can always hit the "shutdown" button, because, after all, the specification is about the nuclear power plant, not the user (in real life the user can obviously always hit the button), we should realize that we are reasoning about a formal model of both the user and the power plant. Hence, verifying such



(a): Scopes (b): Behaviors

Figure 5.3: Pattern Extensions incorporated in PASS

properties can provide important sanity checks on the specification.

Models usually abstract away from the underlying OS scheduling policy, containing no information on how often a process is executed, or how the next process to be executed is chosen; the scheduler is treated non-deterministically. The effect of this choice is that there can be traces in the LTS representation of the model that do not correspond to realistic scenarios, for example, runs in which a single process gets all execution time, while other processes starve. If a process never gets a chance to run, it usually cannot reach its goal. Unless the scheduler is explicitly modeled, we often want to disregard such unrealistic loops, since they may lead to false-negatives: unrealistic counter-examples when model checking liveness properties. This is in particular relevant for the *liveness* category of properties, which require that something good eventually happens. To disregard such behavior, we have added *fairness* variants of the *existence* and *response* patterns, which rely on the assumption that in reality each process of the system is given a fair chance to execute. It is important to note that fairness is not a property to be verified, but rather, a constraint or an assumption that can be expressed in temporal logic. Such a property will be evaluated only along fair paths (with respect to the action/state captured by the property). The action-based formulation of fairness in the context of the modal μ-calculus extended with data has already been mentioned in the early work of [83]. Specifically, an example formula is presented to express that "every datum d_0 which is inserted in a queue by an action $r(d_0)$ will be eventually delivered by an action $s(d_0)$, taking into account only the execution paths that are *fair* w.r.t. the action $s(d_0)$, i.e., those paths which cannot infinitely often enable $s(d_0)$ without infinitely often executing it." This is in fact the basis for the *response under fairness* pattern and its scope variations.

Inspired by [51], further variants of the *precedence* and *response* patterns have also been added to PASS. For example, consider the following property:

Every request for the resource is eventually acknowledged. Also, a request must eventually arrive in a correctly functioning system.

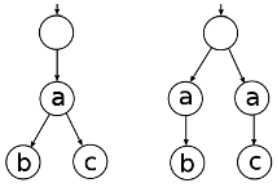


Figure 5.4: Branching versus linear time Logic

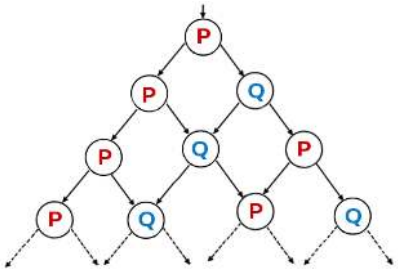


Figure 5.5: Branching time logic can capture possibility properties

There is an additional constraint to this *response* property, namely, that the cause event must take place. This can also be seen simply as a conjunction of two properties which should hold simultaneously, but for convenience reasons it was added as a separate pattern. Taking into account a similar consideration, a *precedence* variant is added. All behavior extensions are shown in Figure 5.3b. Adding 4 scope and 6 behavior variations have led to more than 150 ((5+**4**)∗(11+**6**)) new unique patterns to be chosen from [164].

Given that the pattern system is necessarily abstract, and there are mappings (or implementations) in temporal logics with different expressive powers, a natural question arises: are there patterns that cannot be expressed if we had chosen another logic (such as LTL or CTL), or patterns that, when formulated in a different logic, yield formulas with different meanings? For example, both Kripke structures in Figure 5.4 have the same runs from the perspective of LTL, so every LTL formula will have the same truth value in both. A CTL formula, however, can make the distinction. For example, a property saying *a must happen, and immediately afterwards, b may happen and c may happen* will be true for the left structure, but false for the one on the right, when specified with CTL. On the other hand, a pure branching time logic like CTL is not expressive enough for capturing the *fairness* pattern extensions. A more powerful logic like CTL* [70] or modal μ-calculus is necessary. LTL can express fairness constraints; however, *possibility* properties (our *always-enabled* pattern extension) cannot be formulated, while they can in CTL. For example, let P denote the predicate "transition p is executed" (similar for Q); then every subtree of the unfolding Kripke structure in Figure 5.5 satisfies the property *P is always possible*, but LTL cannot capture this, because it is only looking at linear paths, not computation trees. In conclusion, linear and branching temporal logic are incomparable. Even within each world, there are logic variations which add expressive power at the cost of computational complexity of the verification[3]. The modal μ-calculus, while

[3]CTL model checking has a (theoretical) polynomial complexity with respect to the size of the specification, while LTL model checking is exponential. However, many experimental results show that in practice, for certain categories of systems and formulas, the relative merits of CTL are lost. We omit further discussion on the topic.

considered less intuitive compared to the more commonly used ones (like LTL and CTL), is able to capture certain property patterns which either LTL or CTL cannot.

5.4.3 Sequence Diagrams for Visual Property Specification

In our experience, visual scenarios are often the most suitable and commonly used means to understand the dynamics of a system where a multitude of communicating components deliver the intended behavior. We believe that such a visual depiction of scenarios, more than finite state machines, improves the practitioner's understanding of requirements as well. This is why we chose sequence diagrams as a property specification artifact too. A controlled experiment study [6] has also concluded that the presence of sequence diagrams significantly improves the comprehension of the modeled functional requirements.

There are also two conformance-related combined fragments that we mentioned in Chapter 4: *assert* and *negate*. Their use in practice is limited, because their semantics described in the UML 2.x specification [92] are somewhat vague and confusing. By default, sequence diagrams without the use of these two operators only reflect possible behavior, while *assert* and *negate* alter the way a trace can be classified as valid or invalid. The potential problems with the UML 2.x assertion and negation are explained in [100]. In summary, the specification aims at depicting required and forbidden behaviors. However, as [100] points out, stating that "the sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace" suggests that the invalid set of traces for an *assert* fragment is its complement, i.e., the set of all other possible traces. Conversely, the standard also declares that the invalid set of traces are associated only with the use of a *negate* fragment, which is contradictory with the previous statement. These two operators should be considered as truth modalities, and we assign the following semantics: *negate* is considered a set-complement operator for the event captured by the fragment, while *assert* specifies that an event must occur. For the reasons described above, we disallow nesting between these two fragments. We find that this does not limit the expressiveness of property specifications in practice.

Most of the invented notations used by existing scenario approaches can fit well in UML 2.x sequence diagrams. By creating a simple UML profile, we used the stereotypes mechanism to apply the restrictions on the usage of *negate* and *assert*, as well as to distinguish events representing pattern scope interval bounds from regular ones. As an example, Figure 5.6 depicts the *precedence chain* pattern (with a *between-Q-and-R* scope), with the stereotypes applied to messages Q and R. The pattern expresses that event P must precede the chain of events S, T, always when the system execution is in the scope between events Q and R. Notice that we do not have to specify additional textual restrictions on past and future events, nor

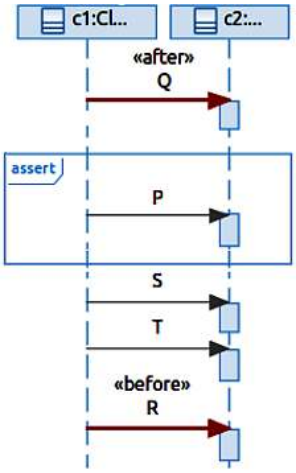


Figure 5.6: UML2 Sequence Diagram with applied stereotypes

transform the scenario in a negative form by introducing an *error* message. They are automatically reflected in the μ-calculus formula, as long as there is a distinction between interval-marking messages, regular, mandatory, and forbidden ones.

5.4.4 Transforming a μ-calculus Formula Into a Monitor Process

In mCRL2, verification of μ-calculus formulas is conducted using tooling that operates on systems of fixpoint equations over first-order logic expressions. This sometimes requires too much overhead to serve as a basis for lightweight bug-hunting, as it can be difficult to interpret the counter-examples that are obtained from these equation systems in terms of the original mCRL2 process. Observers, or monitors (à la Büchi) defined in the mCRL2 model itself, can sometimes be used to turn the model checking problem into a reachability one. A general model checking mechanism used with tools like SPIN is to construct a Büchi automaton for an LTL formula, which accepts exactly those executions that violate the property. A product of the model state space (typically a Kripke structure) and the Büchi automaton is then composed, and checked for emptiness [62]. Although syntactically Büchi is similar to the finite-state monitor for which we aim, the difference lies in the acceptance conditions: a monitor accepts only finite runs of the system, while Büchi can trap infinite executions through detection of cycles, but potentially needs the entire state space generated in the process. Runtime verification does not store the entire state space of a model, so it cannot detect such cycles. In addition, to expose state information, the transitions in Figure 5.7 are labeled with elementary propositions that hold in the target states, rather than actions (notice the $\wedge$ operator, it does not refer to two actions occurring simultaneously). As such, we cannot directly use and adapt the existing tools and algorithms for constructing Büchi automata, within our

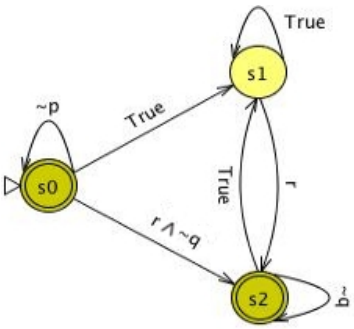


Figure 5.7: A Büchi automaton

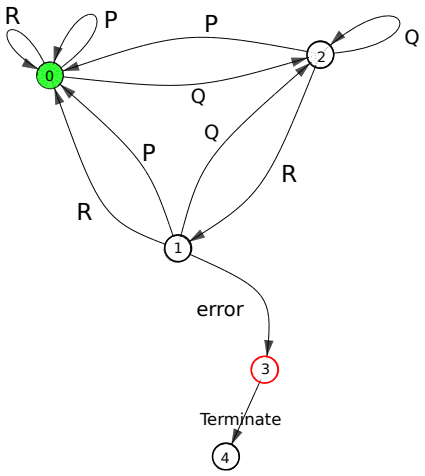


Figure 5.8: Transforming a μ-calculus formula into a monitor

approach.

Not every property can be monitored at runtime when only a finite run has been observed so far. Monitorable properties are those for which a violation occurs along a finite execution, since these monitors have a limited "memory". This problem has been studied [21], and it is known that the class of monitorable properties is strictly larger than that of *safety* properties. In particular, the definition of *liveness* requires that any finite system execution must be extendable to an infinite one that satisfies the property. However, by defining an end-scope of a property, we can also assert violations to *existence* and *response* patterns, which are typically in the *liveness* category. Such a runtime monitor can also assert *universality*, *absence* and *precedence* patterns with or without scope combinations. We found that we are able to construct a monitor for about 50% of the property patterns covered by PASS.

We translate a core fragment of the μ-calculus to mCRL2 processes which can subsequently serve as observer processes for monitorable properties. The idea behind the translation is that any violation of a property of the form $\forall d{:}D.[R(d)]b(d)$, where b is a simple Boolean expression, is due to the existence of a sequence of the form $R(d)$ for some d of type D. A monitor can then be obtained by converting the regular expression R to an mCRL2 process which represents the non-deterministic finite automaton accepting that regular expression (see [110]). Without loss of generality, we restrict to the following μ-calculus grammar for the class of monitorable properties:

$$
\begin{array}{ll}
\phi & ::= b \mid \forall d:D.\phi_1 \mid \phi_1 \wedge \phi_2 \mid [R]\phi \mid [R^*]\phi \\
R_1, R_2 & ::= \alpha \mid R_1 \cdot R_2 \mid R_1 + R_2 \\
\alpha_1, \alpha_2 & ::= b \mid \mathsf{a}(\mathsf{e}) \mid \neg\alpha \mid \alpha_1 \cap \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \exists d : D.\alpha
\end{array}
$$

The translation proceeds as follows. The function TrS takes two arguments (a formula and a list of typed variables) and produces a process. It is defined inductively as follows:

$$\begin{aligned}
&\mathsf{TrS}_l(b) && = (\neg b \to \mathsf{error}) && (1)\\
&\mathsf{TrS}_l(\forall d{:}D.\phi_1) && = \textstyle\sum d{:}D.\mathsf{TrS}_{l +\!\!+ [d:D]}(\phi_1) && (2)\\
&\mathsf{TrS}_l(\phi_1 \wedge \phi_2) && = \mathsf{TrS}_l(\phi_1) + \mathsf{TrS}_l(\phi_2) && (3)\\
&\mathsf{TrS}_l([R]\phi) && = \mathsf{TrR}_l(R) \cdot \mathsf{TrS}_l(\phi) && (4)\\
&\mathsf{TrS}_l([R^*]\phi) && = \mathsf{TrS}_l(\phi) + X(l) \cdot \mathsf{TrS}_l(\phi) && \\
& && \textit{where } X(l) = \mathsf{TrR}_l(R) \cdot X(l) + \mathsf{TrR}_l(R) \textit{ is a fresh recursive process} && (5)
\end{aligned}$$

where TrR takes a regular expression (and a list of typed variables) and produces a process or a condition:

$$\begin{aligned}
&\mathsf{TrR}_l(\alpha) && = \bigoplus_{a \in Act} (\textstyle\sum d_a{:}D_a.\ \mathsf{Cond}_l(a(d_a), \alpha) \to a(d_a)) && (6)\\
&\mathsf{TrR}_l(R_1 \cdot R_2) && = \mathsf{TrR}_l(R_1) \cdot \mathsf{TrR}_l(R_2) && (7)\\
&\mathsf{TrR}_l(R_1 + R_2) && = \mathsf{TrR}_l(R_1) + \mathsf{TrR}_l(R_2) && (8)
\end{aligned}$$

where $\bigoplus$ is a finite summation over all action names $a \in Act$ of the mCRL2 process, and where Cond takes an action and an action formula and produces a condition that describes when the action is among the set of actions described by the action formula α:

$$\begin{aligned}
&\mathsf{Cond}_l(a(d_a), b) && = b && (9)\\
&\mathsf{Cond}_l(a(d_a), a'(e)) && = \begin{cases} d_a = e & \text{if a = a'} \\ \text{false} & \text{otherwise} \end{cases} && (10)\\
&\mathsf{Cond}_l(a(d_a), \neg\alpha_1) && = \neg\mathsf{Cond}_l(a(d_a), \alpha_1) && (11)\\
&\mathsf{Cond}_l(a(d_a), \alpha_1 \cap \alpha_2) && = \mathsf{Cond}_l(a(d_a), \alpha_1) \wedge \mathsf{Cond}_l(a(d_a), \alpha_2) && (12)\\
&\mathsf{Cond}_l(a(d_a), \alpha_1 \cup \alpha_2) && = \mathsf{Cond}_l(a(d_a), \alpha_1) \vee \mathsf{Cond}_l(a(d_a), \alpha_2) && (13)\\
&\mathsf{Cond}_l(a(d_a), \exists d{:}D.\alpha_1) && = \exists d{:}D.\mathsf{Cond}_l(a(d_a), \alpha_1) && (14)
\end{aligned}$$

In Appendix A.1 we formally define the syntactic elements of mCRL2 and the associated semantics. We also provide the formal semantics of modal μ-calculus formulas. By providing such mapping of the syntactic constructs to a mathematical domain, we have the necessary grounds to prove the monitor construction correctness. We formulate the following lemmas informally at this point, in order to make them more accessible to readers; more formal definitions, as well as proofs can be found in Appendix A.2:

Lemma 1. A (multi-)action $a(d)$ belongs to the set of actions described by α iff $\mathsf{Cond}_l(a(v), \alpha)$ holds for a variable v having value d.

Lemma 2. A path (or a word) ρ is accepted by the regular expression characterized by R iff the constructed process $\mathsf{TrR}_l(R)$ can terminate successfully after performing the sequence of actions in ρ.

Lemma 3. A path ρ is accepted by the regular expression characterized by R^+ iff the constructed recursive process $X(l) = \mathsf{TrR}_l(R) \cdot X(l) + \mathsf{TrR}_l(R)$ can terminate successfully after performing the sequence of actions in ρ.

Lemma 4. A process P violates the formula ϕ iff the synchronous parallel composition of processes P and $TrS_l(\phi)$ can reach a state in which the action *error* is enabled.

Using the above translation, our tool automatically constructs such a monitor which is placed in parallel with the system model, to perform runtime verification. Clearly, in the "worst" case, if the model is correct with respect to the property, all relevant model states will be traversed. In practice however, refutation can often be found quickly after a limited exploration.

5.4.5 A Walk-Through Example

To demonstrate how a μ-calculus formula is encoded in an mCRL2 monitor process in practice, we use one of the simpler patterns as a walk-through example: the *existence* behavior with a *between-Q-and-R* scope. The formula with generic placeholders P, Q, and R, is: $[true^*.\ Q.\ (\neg(P \cup R))^*.\ R]\ false$
A step-by-step translation applied on a model with general actions *action_1*, *action_2*,..., *action_n*, is given below. For clarity, the identifiers of the constructed processes are kept descriptive, they correspond to (segments of) the formulas being translated between quotes. The translation rules, shown next to each process, are applied recursively until all monitor processes have been resolved. As an initial step, the formula is converted in an equivalent form, more suitable for conversion[4]:

$$proc\ Monitor\ =\ Mon\_"[true^*][Q][(\neg(P \cup R))^*][R]\ false";$$

$$proc\ Mon\_"[true^*][Q][(\neg(P \cup R))^*][R]\ false" = Mon\_"[Q][(\neg(P \cup R))^*][R]\ false" + Mon\_"X_1" \cdot Mon\_"[Q][(\neg(P \cup R))^*][R]\ false"; \quad \ldots\ rule(5)$$

$$proc\ Mon\_"X_1" =\ Mon\_"true" \cdot Mon\_"X_1" + Mon\_"true"; \quad \ldots\ rule(5)$$

$$proc\ Mon\_"true" =\ \textstyle\sum_n(true)\ \rightarrow\ action\_n\ ; \quad \ldots\ rule(6), (9)$$

[4]Observing the identity $[R_1 \cdot R_2]\phi = [R_1][R_2]\phi = [R_1]\phi_1$, where $\phi_1 = [R_2]\phi$

$$proc\ Mon\_\text{“}[Q][(\neg(P \cup R))^*][R]\ false\text{”} = Mon\_\text{“}Q\text{”} \cdot Mon\_\text{“}[(\neg(P \cup R))^*][R]\ false\text{”}; \quad \ldots\ rule(4)$$

$$proc\ Mon\_\text{“}Q\text{”} = \sum_n (action\_n == Q) \ \rightarrow\ action\_n\ ; \quad \ldots\ rule(6), (10)$$

$$proc\ Mon\_\text{“}[(\neg(P \cup R))^*][R]\ false\text{”} = Mon\_\text{“}[R]\ false\text{”} + Mon\_\text{“}X_2\text{”} \cdot Mon\_\text{“}[R]\ false\text{”}; \quad \ldots\ rule(5)$$

$$proc\ Mon\_\text{“}X_2\text{”} = Mon\_\text{“}(\neg(P \cup R))\text{”} \cdot Mon\_\text{“}X_2\text{”} + Mon\_\text{“}(\neg(P \cup R))\text{”}; \quad \ldots\ rule(5)$$

$$proc\ Mon\_\text{“}(\neg(P \cup R))\text{”} = \sum_n (\neg((action\_n == P) \ \vee\ (action\_n == R))) \ \rightarrow\ action\_n\ ; \quad \ldots rule(6), (11), (13)$$

$$proc\ Mon\_\text{“}[R]\ false\text{”} = Mon\_\text{“}R\text{”} \cdot Mon\_\text{“}false\text{”}; \quad \ldots\ rule(4)$$

$$proc\ Mon\_\text{“}R\text{”} = \sum_n (action\_n == R) \ \rightarrow\ action\_n\ ; \quad \ldots\ rule(6), (10)$$

$$proc\ Mon\_\text{“}false\text{”} = \textbf{\textit{error}}; \quad \ldots\ rule(1)$$

The resulting state space of the monitoring processes is visually depicted in Figure 5.8. Intuitively, the monitor process will step through those exact actions that the original system takes. If a sequence of steps refuting the formula is completed, the monitor will execute the “error” action as a last step, indicating that a counter-example trace has been found. More examples of monitors along with references to the applied transformation rules in each step, can be found at [163].

5.4.6 PASS Integration in the Eclipse Platform

We chose to stay in IBM’s RSA environment which we used for the model transformation methodology presented in Chapter 4. One of the advantages is that RSA is built on top of Eclipse and the GMF (Graphical Modeling Framework), making it relatively easy to extend the functionality. PASS is developed as an Eclipse plug-in (Figure 5.9), using a lightweight UML profile for the stereotypes defined in Section 5.4.3. The Eclipse MDT-UML2 plug-in provides an EMF-based Java implementation of the UML 2.x metamodel for Eclipse. Our plug-in uses it for gathering information about the workspace UML model being verified, as well as for generating the sequence diagram representation of a property. Being developed as a conversational tool (a wizard), it relies on the JFace and SWT toolkits for the UI implementation (views/dialogs). ANTLR provides the necessary lexical analyzer and

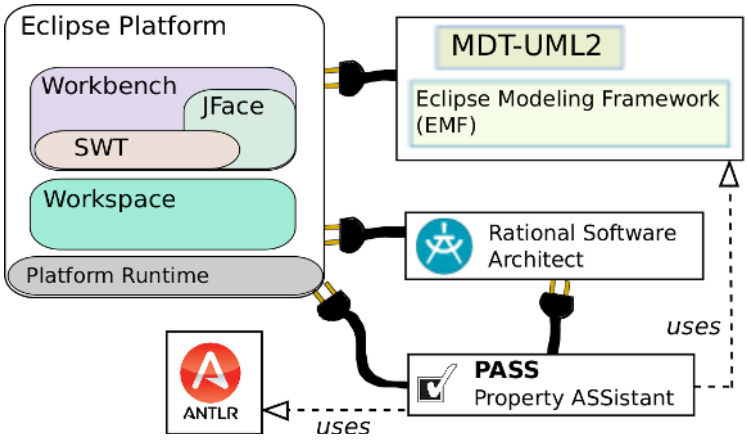


Figure 5.9: PASS integration in Eclipse

parser generation from the μ-calculus grammar, needed for the monitor synthesis. Using the monitor, the counter-example will be provided at the UML level.

5.5 PASS by Example: Revisiting DIRAC

We demonstrate the applicability of PASS for eliciting properties by revisiting the SMS and WMS modules of DIRAC. We created a UML model of the SMS, together with the communication interfacing the WMS. One of the problems originally discovered with model checking (Section 3.5.3) was the fact that the callback from the SMS was not always properly handled by the WMS. If a job had been manually "Killed" by production managers while it was waiting on a staging request to complete, and the callback from the SMS arrived only later, it would awaken the job and cause it to eventually start running. Such "zombie" jobs were discovered in the system on several occasions, and a simple fix was implemented (mid. 2011) in the WMS to properly guard the callback against this particular case. Reflecting this change in the implementation and the model, it was confirmed that the SMS callback no longer affected the job state if the original "Staging" one had changed to some other status in the meantime. However, another problem appeared later in production. Jobs would remain in a "Staging" state without progress, and any staging request information was apparently already gone from the SMS, making it difficult to investigate the root cause based solely on logging traces analysis. To capture the underlying problem formally for the purpose of model checking, we established the following informal property statement: *any job requesting staging must eventually make progress (the job must be called-back properly) before the corresponding task is removed from the SMS.*

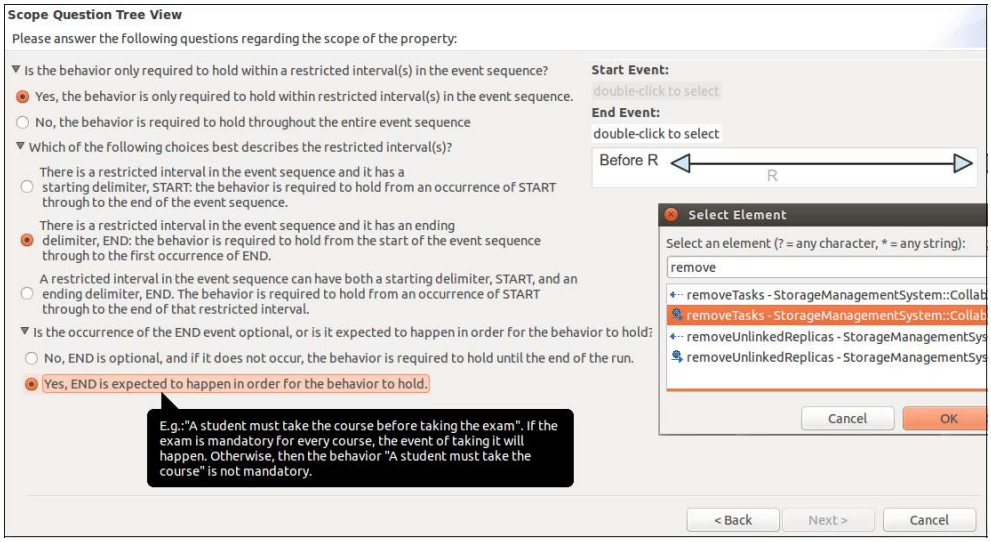


Figure 5.10: Eliciting the scope for a property with PASS

5.5.1 PASS: The Property ASSistant

To cope with the ambiguity of system requirements, PASS guides the practitioner through a series of questions to distinguish the types of scope and behavior as a relation between multiple events. By answering these questions, one is led to consider some subtle aspects of the property, which are typically overlooked when manually specifying the requirement in temporal logic. Having recognized the last part of the property (i.e., "*before the corresponding task is removed from the SMS*") as a scope restriction, we need to choose the appropriate answers from the scope question tree (shown in Figure 5.10). With respect to the last question, since the behavior of the SMS agents is cyclic (should not terminate in practice), and we expect that the event of task removal happens, according to the original problem description, the second answer is chosen. A hint with an additional explanation is also displayed in a popup box next to the selected answer, to clarify its meaning. The actual communication event can be selected by double-clicking the end-event placeholder. This presents the user with a pop-up window with a list of all the message exchanges in the model, so the appropriate one can be chosen, in this case the *removeTasks* message.

To elicit the behavior part of the property ("*any job requesting staging must eventually make progress, i.e., the job must be called-back properly*"), some clarification is needed. There are two possible outcomes of staging: it can either succeed, in which case the job is called back with "Done", or fail, in which case the job is called back with a "Failed" status. Therefore, we need to break down the property elicitation in two separate cases, both of which should hold. We treat them separately, and

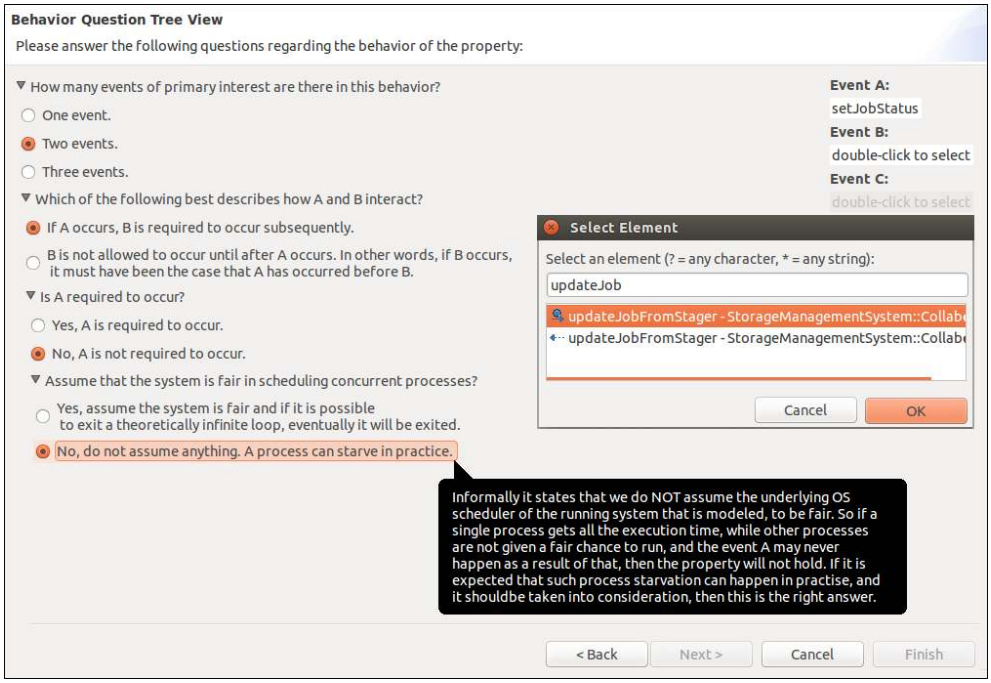


Figure 5.11: Eliciting the behavior for a property with PASS

only elaborate the former case. The Behavior Question Tree part of the wizard is shown in Figure 5.11. While scope elicitation typically involves identifying words like “before”, “after”, “until”, and “between”, the relevant behavior actions are decided by focusing on the verbs in the informal property description. In this case, the two events are (1) setting the job in a “Staging” state, and (2) issuing a successful callback to the job. The selected answers reflect the fact that these two events have a causal relationship, and not every job requires staging. At the end of the questionnaire, the user is presented with a natural language summary (Figure 5.12) of the requested property, which can be reviewed before making the final decision. A μ-calculus formula pertaining to the property is presented, along with the possibility to assign concrete parameter values that the messages carry. The input to our model is reduced to the minimum that retains the behavioral characteristics, so a single job with one requested file was considered sufficient for this property. In addition, a sequence diagram (Figure 5.13) and a monitor process in mCRL2 are generated, to be used in the final model checking phase. The case with failed staging is elicited in an analogous manner.

Model checking the property on the model augmented with the generated monitor revealed a counter-example trace where it does not hold. To understand why, the sequence diagram in Figure 5.14 shows the fix originally applied (the *break* fragment). Some communication details are omitted and messages are enumerated for

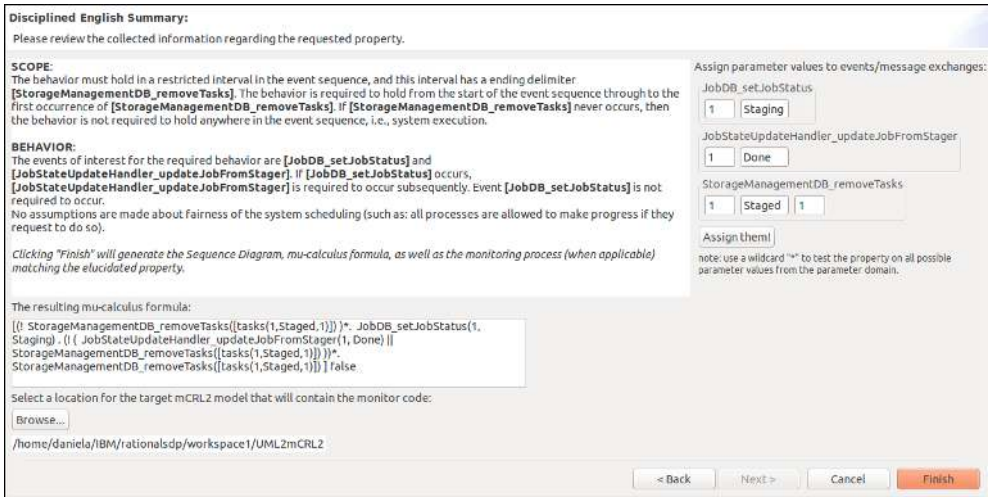


Figure 5.12: Summary of the elicited property with PASS

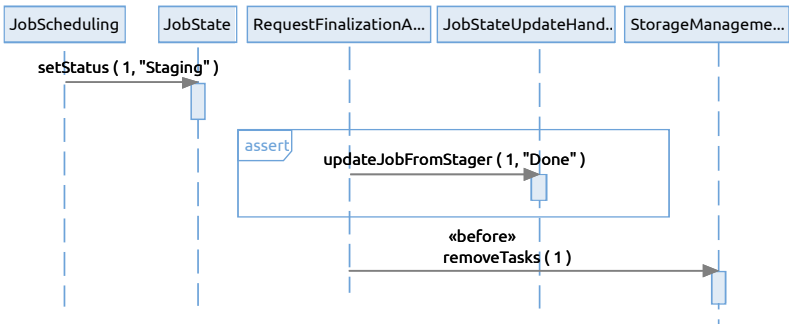


Figure 5.13: A sequence diagram visualizing the elicited SMS property

clarity. Partial temporal ordering between these messages is assumed, as imposed by the UML semantics. This implies that in practice the sequence of events involving the *RequestFinalizationAgent* (beginning with message *7:updateJobFromStager*) can happen before *4:setJobStatus(jobID,"Staging")* takes place. In particular, when the staging request is processed very quickly by the SMS (e.g., if all files are already cached when the request arrives), the callback from the agent may arrive before the job is set to a "Staging" state, and as a consequence, will be ignored because of the applied fix. Such unfortunate jobs will remain in a "Staging" state forever, due to the fact that the request had been processed from the viewpoint of the SMS, and the staging task associated with it has been removed (*11:removeTasks(jobID)*).

Another issue with the SMS was the occasional MySQL error present in the logs, caused by a foreign key constraint violation (log excerpts shown in Figure 5.15). Using knowledge about the database organization, the only way for this foreign constraint violation to manifest is if information referenced by the StageRequests

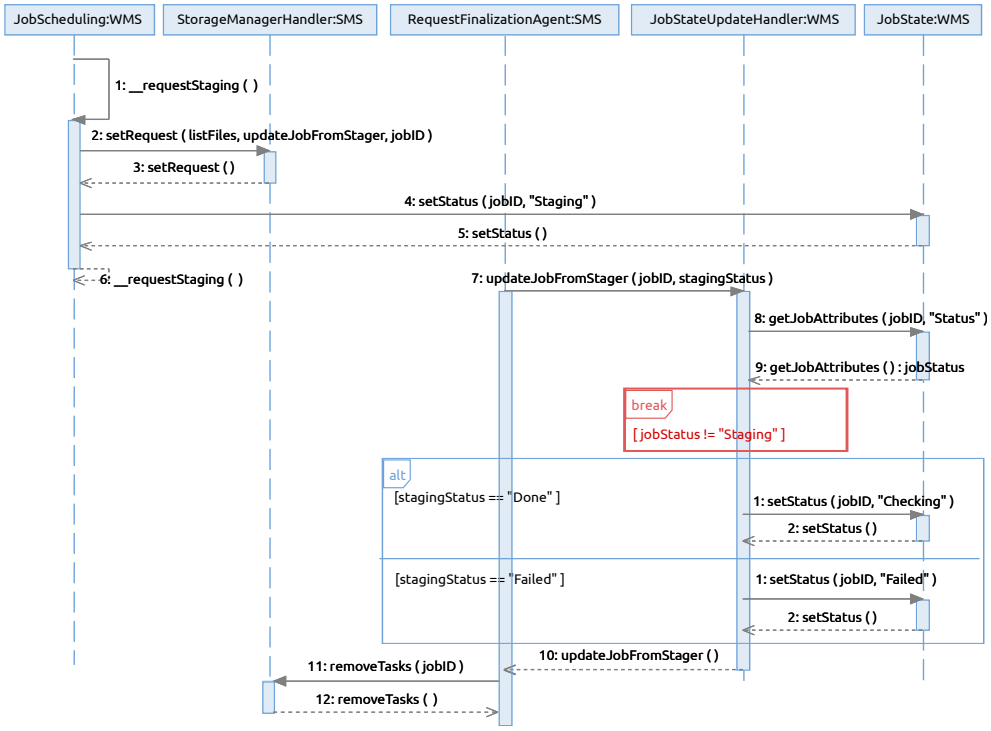


Figure 5.14: Fix applied for the problem with "zombie" jobs

```
2014-07-28 12:15:41 UTC StageRequestAgent ERROR: StageRequest._issuePrestageRequests: Failed to
insert stage request metadata. Execution failed.: ( 1452: Cannot add or update a child row: a
foreign key constraint fails (`StorageManagementDB`.`StageRequests`, CONSTRAINT
`StageRequests_ibfk_1` FOREIGN KEY (`ReplicaID`) REFERENCES `CacheReplicas` (`ReplicaID`)) )

2014-08-04 16:37:34 UTC StageRequestAgent ERROR: StageRequest._issuePrestageRequests: Failed to
insert stage request metadata. Execution failed.: ( 1452: Cannot add or update a child row: a
foreign key constraint fails (`StorageManagementDB`.`StageRequests`, CONSTRAINT
`StageRequests_ibfk_1` FOREIGN KEY (`ReplicaID`) REFERENCES `CacheReplicas` (`ReplicaID`)) )

2014-08-05 09:38:19 UTC StageRequestAgent ERROR: StageRequest._issuePrestageRequests: Failed to
insert stage request metadata. Execution failed.: ( 1452: Cannot add or update a child row: a
foreign key constraint fails (`StorageManagementDB`.`StageRequests`, CONSTRAINT
`StageRequests_ibfk_1` FOREIGN KEY (`ReplicaID`) REFERENCES `CacheReplicas` (`ReplicaID`)) )
```

Figure 5.15: MySQL errors in the SMS logs

entry was removed from the CacheReplicas table prior to submitting a staging request. This, in turn, happens after the task associated with this replica is called back and removed. Thus, it was already assumed that the task had somehow been removed along with the replica information, before a staging request was submitted. However, the circumstances for such behavior were unclear. In principle, *there should not be any staging requests for a task, once it has been called back to the corresponding job*. We can elicit this property with PASS in a similar manner as previously, using the *removeTasks* message as an *after-Q* beginning scope delimiter. The only event

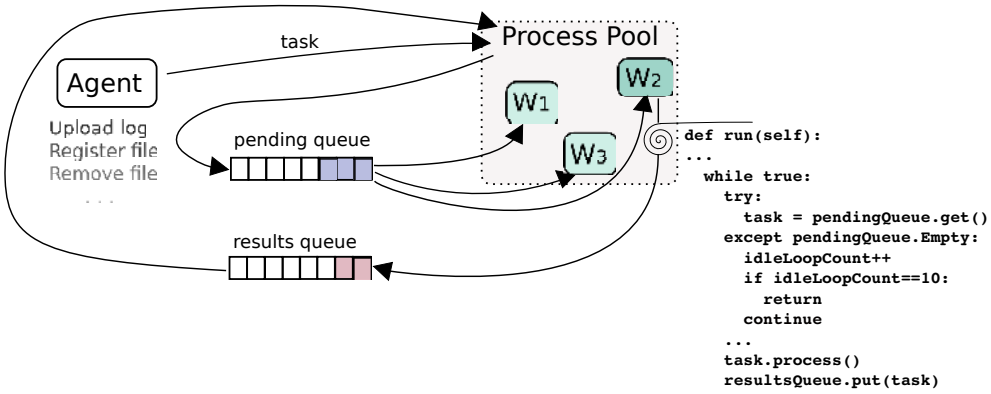


Figure 5.16: DIRAC ProcessPool

of interest for the behavior part of the property is the *insertStageRequest* call, which should be absent for this particular task. Checking the resulting model augmented with the generated monitor did not reveal any misbehavior. However, changing the UML model simply by extending the input to a single job, but requesting two files rather than one, already exposed the problem. The counter-example indicated a typical race condition. Failing to stage one file (for whatever reason) will already fail the whole task and eventually trigger the Request Finalization Agent to callback the corresponding job and clear the task information. On the other hand, just before this happens, the Stage Request Agent can select the other waiting file eligible for staging. After submitting the actual staging request, the attempt to add information about it in the database will result in a MySQL error, as the task along with the related replica information has been meanwhile removed. Although this is not a major impediment for operations, the number of staging requests accounted for is in effect smaller than the number of submitted ones.

5.5.2 PASSWebStart: The Alternative

While not our primary goal, a standalone Java Web Start version of the tool was also developed, for deriving properties for mCRL2 models created manually and independently of any UML environment. In this context, rather than generating a sequence diagram, for monitorable properties we used Graphviz [79] (already included in most Linux distributions) to provide a visual representation of the state space of the generated monitor. The mCRL2 toolset provides means for converting the state space of a process into DOT, a simple graph description language used as input format by Graphviz. PASSWebStart [161] has the same look and feel as PASS, except that it does not depend on the Eclipse Modeling Framework, but only on the mCRL2 grammar for parsing the input model and obtaining the set of all action names (often called its *alphabet*) defined in it.

5.5.3 The DIRAC ProcessPool

We applied the PASSWebStart tool for eliciting a number of properties on a DIRAC core utility module: the *ProcessPool*. As the name suggests, the ProcessPool is a piece of logic that maintains a pool of concurrently running *worker processes* to handle a queue of tasks, resembling a producers/consumers paradigm. Agents can use the ProcessPool to fill the queue with tasks to be executed, while the worker processes execute these tasks. For example, it is used by a special agent responsible for handling requests for asynchronous operations such as uploading log files from jobs, registering files in a catalog, or removing files from storage. Figure. 5.16 is a simplified illustration of the ProcessPool dynamics, controlled by three configuration parameters: minimum and maximum number of workers that can be alive at any given moment, and a maximum number of queued tasks in the *pending queue*. The number of active workers is automatically increased and decreased depending on the current load, of course not exceeding the minimum/maximum limits. New workers are only created when the pending queue is not empty, and there are no free existing workers to process the queued tasks. This elacticity is achieved in a separate background (daemon) thread of the ProcessPool. This background thread is also responsible for picking up the results from the *results queue* and issuing callbacks to the agent. An attempt to add a task to a full pending queue will result in a blocked execution, until a task is dequeued by some worker process instance, which subsequently executes the actual task.

The main thread of every worker process executes an "infinite" *while*-loop, reading a task from the pending queue, executing it, and pushing back the result to the results queue. A worker can self-destroy after a certain amount of idling: on every attempt to read from an empty queue, the idle-loop counter is increased, and if this counter reaches a value of 10, the process returns from its *run()* method. Since the queues are shared among concurrent processes, a locking mechanism is put in place to prevent race conditions and data corruption.

Occasionally, the agent using the ProcessPool would become stuck, not showing any progress, so a formal model was created manually to investigate the reasons. The only clues about the possible root cause suggested a pattern where this would happen immediately upon an attempt to enqueue a task, preceded by a certain period of agent idleness without any new incoming requests. To capture some aspects crucial for the correct functioning of the ProcessPool, the following properties were formulated and elicited with the tool:

1. The system is free of deadlocks.

2. If a task is pushed in the pending queue, it will eventually end up in the results queue.

Table 5.2: Formulas obtained with PASSWebStart

Property	μ-calculus formula
PR1	$[true^*]\langle true\rangle true$
PR2	$[true^*.pendingQueuePut(task(1, false, LogUpload))]\mu X.\langle true\rangle true \wedge [\neg\ resultsQueuePut(task(1, true, LogUpload))]X$
PR3	$[(\neg\ pendingQueuePut(task(1, false, LogUpload)))^*.$ $pendingQueuePut(task(1, false, LogUpload)).$ $(\neg\ resultsQueuePut(task(1, true, LogUpload)))^*.$ $resultsQueuePut(task(1, true, LogUpload)).$ $(\neg\ resultsQueuePut(task(1, true, LogUpload)))^*.$ $resultsQueuePut(task(1, true, LogUpload))]\ false$
PR4	$[true^*]\ [prListLock\_release.\ (\neg\ prListLock\_acquire)^*]$ $[\forall arg1{:}Task.pendingQueuePut(arg1)]\ false$
PR5	$[true^*]\ \mu X.\langle true\rangle true \wedge [\neg prListLock\_acquire]X$
PR6	$[true^*.\ prListLock\_acquire.\ (\neg\ prListLock\_release)^*]$ $\langle true^*.\ prListLock\_release\rangle true$
PR7	$[true^*.prListLock\_acquire.\ (\neg\ prListLock\_release)^*.$ $prListLock\_acquire]\ false$

3. If a task is pushed in the pending queue, it will end up at most once in the results queue.

4. Between releasing a lock and acquiring it again, the pending queue should not be filled.

5. Always, acquiring a lock will eventually be possible.

6. On all fair runs, every acquired lock will eventually be released.

7. A critical section can only be accessed by one process at a time.

For an impression of their complexity, even for these relatively simple properties, the derived formulas are listed in Table 5.2.

Faced with state-space explosion on a typical desktop configuration, for verification purposes we resorted to using a more powerful Intel(R) Xeon(R) @2.27GHz

machine with 70GB RAM. Model checking exposed a deadlock, and quite expectedly, the liveness properties were also refuted. Table 5.3 shows the results and the resource usage for each property. Not all properties are amenable to using the monitor mechanism, and (save for the deadlock search) verification times span about two days in such cases where a formula must be used directly. Notably, model checking the 3rd and 4th property leads to a quicker answer, but requires significantly more memory when using a monitor. This effectively means that the synchronous parallel composition of the original model with the monitor yields a larger state space in these cases, which must be fully explored when the property holds, i.e., the search for the *error* action is unsuccessful. The 70GB of memory were unfortunately not sufficient for verifying the last property by using a monitor.

The deadlock trace exposes a very specific interplay between the components. In a situation with a single idle worker and no tasks in the pending queue, the worker would eventually give up polling the empty queue and return from its main *run()* method. Between these two atomic steps (the last increase of the *idleLoopCount* counter and executing a *return*), it is perfectly possible for any number of other actions to take place, in an untimed process algebra model. For instance, the pending queue can be filled completely. To see how this is possible, the situation is shown in Figure 5.17. The agent will continue sending requests (tasks) to the ProcessPool, and these will be queued without creating new workers, since there is already one which is still accounted as idle. Once the queue becomes full and the lock is acquired by the ProcessPool, the last attempt to enqueue a task will be a blocking one, until space is freed in the queue. Unfortunately, the only idle worker who can dequeue a task has already decided to self-destruct. Before attempting to create new workers in the pool, the background thread checks the current number of idle workers and the size of the pending queue. However, there is a lock around this check, to serialize access to the list of workers maintained by the ProcessPool. On the other hand, this same lock is already held by the ProcessPool in the last queue-

Table 5.3: Model checking results for the ProcessPool properties

	Monitor		**Formula**		**Property**
	Wallclock	Memory	Wallclock	Memory	**holds?**
PR1	Not monitorable		1m9s	136MB	No
PR2	Not monitorable		3169m44s	13.1GB	No
PR3	606m45s	55.4GB	3904m3s	8.1GB	Yes
PR4	924m47s	55.4GB	2705m18s	8.1GB	Yes
PR5	Not monitorable		3875m40s	21GB	No
PR6	Not monitorable		3139m46s	13.1GB	No
PR7	–	–	1670m31s	5GB	Yes

component	action	state
Worker$_1$	idleLoopCounter=10	pendingQueue workers lock t_1 ⋯ W_1 free
ProcessPool	lock.acquire()	pendingQueue workers lock t_1 ⋯ W_1 acquired
ProcessPool	pendingQueue.put(t_1)	pendingQueue workers lock t_1 ⋯ W_1 acquired
ProcessPool	lock.release()	pendingQueue workers lock t_1 ⋯ W_1 free
ProcessPool	. . .	pendingQueue workers lock t_1 t_2 ⋯ t_m W_1
Worker$_1$	return	pendingQueue workers lock t_1 t_2 ⋯ t_m ⊠
ProcessPool	pendingQueue.put(t_{m+1})	pendingQueue workers lock t_1 t_2 ⋯ t_m ⊠ acquired
daemon process	pendingQueue.empty()	pendingQueue workers lock t_1 t_2 ⋯ t_m ⊠ acquired
daemon process	spawnWorkingProcess()	pendingQueue workers lock t_1 t_2 ⋯ t_m ⊠ acquired
daemon process	lock.acquire()	pendingQueue workers lock t_1 t_2 ⋯ t_m ⊠ acquired
	deadlock	

Figure 5.17: ProcessPool deadlock trace

ing attempt. In effect, the attempt of the background thread to get the number of idle workers (and eventually decide to create new ones if necessary) produces a deadlock, as it can no longer acquire the lock.

One may argue that chronologically this is unlikely to happen in reality. Such a scenario becomes more plausible when considering the low-level Python library implementation of the *multiprocessing.Process* class, which is used for subclassing the worker. Upon returning from its main execution method *run()*, the background machinery takes some steps to cleanup the execution context and update the status of the object, to reflect the process termination. Furthermore, the queues are instances of *multiprocessing.Queue*, whose *.empty()* method is unreliable in a multi-threading environment, as it takes some time for the buffers to be flushed from the underlying pipe into the queues. This method is used by the background thread for making decisions based on the number of tasks in the pending queue.

5.6 Property Specifications with μ-calculus: A Survey of Published Works

As a form of external validation of the PSP classification (and our extension thereof) applicability, we examined 25 published works[5] that use the event-based μ-calculus as a target formalism to express system properties from various domains. Most of them were taken from the CADP and mCRL2 case studies archives, both available online [112; 68]. Our main objective was to reassess the usefulness of the PSP classification in the context of μ-calculus and event-based systems, by fitting each property in a specific pattern. In absence of the original models for many of the publications, the "patterning" was done manually, consulting only the information available in them. After identifying the pattern that should be applied, the formula was compared with the original one used for the property. A secondary objective was to get an indication of how error-prone the manual formalization of properties can be in practice, even when experienced users are involved. Verifying the properties on the actual models was beyond the scope of this work.

5.6.1 Is the PSP Classification Useful?

Not all requirements could be fit in the classification. We classify a requirement as "patternizable" if its semantics can be captured by some pattern from our extended pattern classification. The majority of requirements could be fit into exactly one pattern. However, for some requirements, a conjunction of two patterns was necessary to capture the semantics of the original requirement description. In such cases, individually checking that they all hold is equivalent to using the μ-calculus logical connective *and* ($\wedge$) to obtain a composite formula. We still consider such requirements as patternizable. Furthermore, in some cases it was necessary to reformulate the given requirement description (without loss of meaning) in a way that captures the same semantics, but makes the pattern-fitting more obvious. Consider the following property from [135], where a drilling unit's behavior is verified. The unit consists of a turning table (rotating clockwise) that transports metallic products to a drilling and a testing slot, in a streamlined fashion, one by one:

After the input of a product and a rotation of the table, the main controller cannot command a drilling before the clamp has been blocked.

At first glance, classifying it as an *absence* behavior (*"cannot command a drilling"*) with a *between-Q-and-R* scope will not capture the precise semantics, because the beginning scope delimiter has only one event placeholder, while the requirement has two successive events (*"After the input of a product and a rotation of the table"*). With a slight reformulation:

[5]The list of publications is available at http://remenska.github.io

After the input of a product and a rotation of the table, the clamp must be blocked before the main controller can command a drilling.

This is in fact a *response chain* behavior pattern with a *before-R* scope. Indeed, the formula used in [135] corresponds to this exact pattern.

The results from the survey are shown in Table 5.4. From a total of 178 property descriptions encountered in the publications, around 70% could be fitted within the standard PSP classification. If our extensions are taken into account, the coverage is improved by about 10%. Among the 142 patternizable properties, there were 15 for which a conjunction of two patterns was necessary. The properties that could not be captured by any patterns were usually such that a disjunction (set) of events must be considered instead of a single event, within a pattern placeholder. This is a current limitation of PASS, but enhancing the patterns in this direction is feasible.

The distribution of the patterns is shown in Figure 5.18. An immediate observation, consistent with surveys focusing on state-based systems, is that a rather small

Table 5.4: Survey results: property specifications with μ-calculus

Patternizable?			
No	Yes (PSP)	Yes (Extensions)	**Total**
36 ($\approx$ 20%)	126 ($\approx$ 70%)	142 ($\approx$ 80%)	178

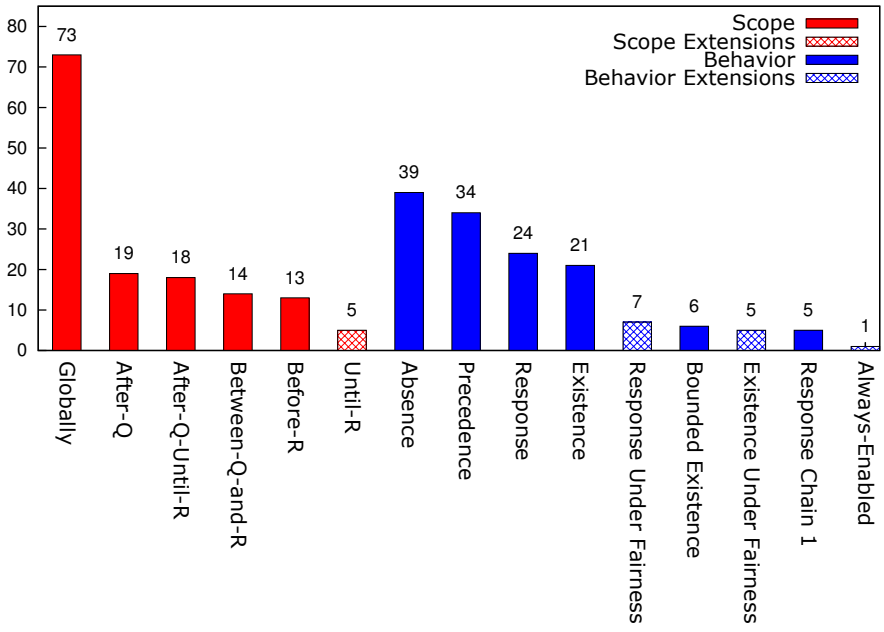


Figure 5.18: Survey of μ-calculus specifications: patterns distribution

number of patterns are sufficient to express the majority of requirements. In fact, about 50% of the properties can be expressed with two patterns: *absence* and *precedence*, which indicates that safety properties are the most common, if the focus is not on a particular domain. Evidently, the most useful extensions are with respect to the *fairness* assumptions (both for *existence* and *response*), encountered in 12 properties. It may be interesting to note that out of the 5 *response chain* properties, 4 were present in a single publication. The chain patterns are typically reported as the least used, due to their complexity. The *universality* pattern was never used, which is coherent with our observation that this pattern, in its original form, is not really interesting for event-based systems. However, we found that the alternative we provided (*always-enabled*) was used once. The results also indicate that the *global* scope is favored, comprising about 50% of the used scopes. We suspect that this is mostly due to the fact that safety properties typically must hold throughout the entire system execution. Only one of our scope extensions, *until-R*, was captured in 3 publications or 5 properties in total.

5.6.2 Is Manual Formalization of Properties Error-Prone?

We encountered several publications with wrong (or suboptimal) property formalizations, where "wrong" means "not conveying the intention of the written natural language requirement". In the following we focus on these publications, discuss each problematic formula in more depth, and provide the correct one. Where necessary, we copy verbatim the informal requirement description, to avoid misinterpretation. We want to clearly state, however, that we intend no criticism on the publication authors. These findings should only serve as an indication that writing correct formulas is a demanding task, and should be supported by some guidance, to avoid errors.

Verification of an Automated Parking Garage

In [137] the requirements of an automated parking garage model are verified by extending the specification with *error* actions which are fired when a requirement is violated. In that sense the approach resembles the monitor construction we presented in Section 5.4.4. Manually implementing such specification extensions can be tedious, impractical for large models, and often error-prone, since the behavior of the original model is altered in the process of augmenting the specification. Furthermore, as already pointed out, not every type of property can be verified in this manner.

Analysis of a Fair Non-Repudiation Protocol

In [40] the authors design and verify a fair non-repudiation protocol. Repudiation is the denial of a previously uttered statement. Non-repudiation should guarantee that if an agent P sends a message m to agent Q, then P cannot deny having sent the message and Q cannot deny having received it. Fairness is a necessary property for communication protocols, and expresses that when the protocol session has terminated, then Q has received m and evidence of origin if and only if P has received evidence of reception. The research contribution is based on the idea of avoiding the use of session labels for identifying individual session runs while still achieving fairness, among other properties. An agent P performs the action $init_P(k, m)$ when it engages in a protocol session, $terminate_P(k, m)$ when the session is over from P's point of view, and $evidence_P(k, m)$ when it receives an evidence of reception of a message m encrypted with key k. We examine the fairness property, stated and formalized as follows:

Fairness (for P) means that if Q gets its evidence, then so shall P. This property says that P does not terminate in an unfair state for P. But since P will eventually terminate (verified with another property), P will indeed terminate in a fair state.

$$[(\neg evidence_Q(k,m))^*.\ evidence_P(k,m).\ (\neg evidence_Q(k,m))^*.\ terminate_P(k,m)]\ false$$

This formula will refute a trace in which P gets evidence before the session terminates, which is not the intention. The actions $evidence_P$ and $evidence_Q$ should be swapped, as the property should describe "fairness for P", while Q plays the role of receiver (and a possible intruder). Thus, the correct formula should be:

$$[(\neg evidence_P(k,m))^*.\ evidence_Q(k,m).\ (\neg evidence_P(k,m))^*.\ terminate_P(k,m)]\ false$$

meaning that any behavior in which Q gets evidence while P does not, before the session is over from P's point of view ($terminate_P$), should not be allowed.

Verification of a Self-Configuration Protocol for Distributed Applications in the Cloud

The task of automatically configuring distributed applications in the cloud is tackled in [170]. The paper focuses on a self-configuration protocol that is able to configure a distributed application whose interconnected software components are deployed and running on (possibly) different VMs, without any centralized server. To verify that the components' initialization dependencies are respected, i.e., a component cannot be started before the components it depends on, the following example is provided:

$$[true^*.\ startcomp\_C1.\ true^*.\ startcomp\_C2]false$$

If a component C1 is connected through a mandatory client interface to a component C2, we generate the property above meaning that we will never find a sequence where C1 is started

before C2.
Based on the given property description, this is an *absence* behavior pattern (C1 should not be started) with an *until-R* or a *before-R* scope, depending on whether it is assumed that all components should eventually be started. If we do not rely on such an assumption, an *until-R* scope should be used:

$$[(\neg startcomp\_C2)^*.\ startcomp\_C1]\ false$$

The original formula will not refute a run (it will miss a property violation) in which the component C2 is never started. The requirement that *all components are eventually started* is captured with a separate property in the publication. Coincidentally, their verification results report a bug[6] which does not ensure that all components involved in the architecture are eventually started. This bug is hidden in the wrongly specified formula.

Verification of a Dynamic Online Auction

Simulation and formal verification in the domain of business processes and commerce activities has been addressed in [42]. A dynamic online auction system has been formally modeled and subsequently checked for correctness. The focus is on a typical auction protocol, where, once the bidding is open, the duration is fixed, and only one or no winner is chosen at the end. If interested in a particular auction, a buying agent must subscribe to the auctioneer agent, in order to receive a call for proposal. The protocol is modeled with a single auction, and a sale of one item by one seller to n buyers who submit their bids to the auctioneer. A buying agent can request to be removed from the auction by canceling the subscription. In the following, we list the incorrectly specified properties, and provide the correct ones. *The auctioneer can not accept a subscription after the auction ends.* This property is formalized as:

$$\langle true^*.\ auctionFailure\ \cup\ auctionSuccess.\ true^*.\ acceptSubscribe\rangle true$$

The formula seems to capture exactly the opposite, namely, that an auction failure or auction success can eventually be followed by an agent subscription. Even if we assume that this is an unintentional error, and the whole formula was intended as a negation, i.e., specified as:

$$\neg(\langle true^*.\ auctionFailure\ \cup\ auctionSuccess.\ true^*.\ acceptSubscribe\rangle true)$$

it is formalized in a sub-optimal way for model checking. The correct specification is a conjunction of two *absence* patterns with an *after-Q* scope:

$$[(\neg auctionFailure)^*.\ auctionFailure.\ true^*.acceptSubscribe]false$$

and

[6] From the paper, it is not clear which exact property was violated.

$$[(\neg auctionSuccess)^*.\ auctionSuccess.\ true^*.\ acceptSubscribe]false$$

which should both hold individually. A very similar observation can be made about the following property: *An unsubscribed buyer can not receive a call for proposal*. This property is expressed as:

$$\neg(\langle true^*.\ acceptCancelSub\_buyer.\ true^*.\ callForProposal\_buyer\rangle true)$$

Moreover, it is stated that the auction specification supports the subscription and unsubscription of a buyer at any time during the auction. It is not clear whether this can be done multiple times, but if this is the case, the formula above will also mistakenly refute a behavior where after cancellation, a buyer decides to subscribe again. The property hereunder should formalize the requirement that *after a subscription, a buyer can receive a call for proposal*:

$$\langle true^*.\ subscribe\_buyer.\ true^*.\ callForProposal\_buyer\rangle true$$

Instead, the formula specifies that there exists a certain running scenario in which a subscribed buyer will receive a call for proposal. It does not guarantee that a subscribed buyer will eventually receive a call for proposal. Besides, the formula will falsely accept a scenario where a subscribed buyer decides to unsubscribe and then receives a call for proposal, as a valid one. Based on this brief description from the paper, it is rather difficult to elicit the correct property behavior without additional information on the meaning of "can receive a call for proposal". If the intent is that a subscribed buyer should be able to receive a call for proposal at any point after subscription, then the correct formula (*always-enabled* behavior, *after-Q* scope) should be:

$$[(\neg subscribe\_buyer)^*.\ subscribe\_buyer.\ (\neg callForProposal\_buyer)^*]\ \langle callForProposal\_buyer\rangle true$$

If instead, the intent is that a subscribed buyer should eventually be able to receive a call for proposal, then the property (*existence under fairness* behavior, *after-Q* scope) is slightly more relaxed:

$$[(\neg subscribe\_buyer)^*.\ subscribe\_buyer.\ (\neg callForProposal\_buyer)^*]\ \langle true^*.\ callForProposal\_buyer\rangle true$$

Verifying Erlang/OTP Components in μCRL

A model that supports the translation of the Open Telecom Platform finite state machine (FSM) implementation from Erlang to μCRL, a predecessor of mCRL2, is developed in [96]. The approach is experimentally evaluated on two small example case studies, a door with code lock system and a coffee machine, both translated into μCRL models. Subsequently properties are defined and formalized in the μ-calculus, and checked for correctness with CADP. Although property formalization and verification was probably not the primary objective of the

approach, we nevertheless report on the mistakes we encountered there. The actual translation is quite involved and makes extensive use of μCRL custom data types. Stacks are defined in μCRL with push and pop operations for manipulating the current state and data of the modeled FSM. These are encoded in a tuple of the form $tuple(state,\ tuplenil(state\_data))$ and saved on the stack. Synchronous communication is modeled by synchronizing the action pair *send_command* and *receive_command* into a new action *cmd*. The given coffee machine example has three states: selection, payment and removal. Selection allows the user to choose the drink, payment displays the price of the selected one and requires payment. After sufficient coins have been paid, the machine goes to a state remove where the drink is prepared and change is returned. The following (self-describing) actions are defined in the μCRL specification: *display_price*, *pay_coin*, *return_coin* and *remove_cup*. The derived model is checked against the property:

$$[cmd(tuple(selection,\ tuple(cappuccino,\ tuplenil(5))))^*.\\ cmd(tuple(pay,\ tuplenil(5)))^*.(\neg remove\_cup)^*]\langle true^*.\ remove\_cup\rangle true$$

stating that, *when cappuccino is selected and after £5 has been paid, the drink will be prepared*. To better understand the problem with this formula, this is its generic form with placeholders:

$$[Q^*.\ P^*.\ (\neg S)^*]\langle true^*.\ S\rangle true$$

Based on the description, it should be a *response* pattern with an *after-Q* scope. Namely, after the cappuccino is selected, if the correct amount is paid, then the drink will eventually be prepared. The pattern template that captures it (under fairness assumption) is:

$$[(\neg Q)^*.\ Q.\ true^*.\ P.\ (\neg S)^*]\langle true^*.\ S\rangle true$$

Besides the fact that the original formula is wrong, its more problematic aspect is that an action such as *display_price*, which displays the price for a selected drink, is not taken into account even though it should be (the coffee machine FSM also indicates this). In other words, it assumes that nothing should happen between Q (cappuccino selection) and P (payment). The correct one takes this into account via the $true^*$ regular expression which can capture any action. Since a sequence without *display_action* in-between will not happen in practice, the original formula will be trivially (vacuously) satisfied, i.e., potential problems will be hidden, because the pre-condition (cappuccino selection followed *immediately* by payment) is never manifested in the model behavior.

The FSM of the simple door with a code lock system consists of two states, *locked* and *open*, initially set to *locked*, and a system code for opening the door. An external action provides a password which is evaluated, and depending on the outcome, the door is either *open* or a warning message is given and the door remains *locked*. One of the properties is formulated as:

$$\langle true^*.\ cmd(abb).\ (pop\_calls(tuple(s\_locked,\ tuplenil(tuplenil(abc)))))^*.\ warning\_message\rangle true$$

with the intended requirement that *when an incorrect password "abb" is received and the current state is s_locked, the action warning_message will be fairly performed*. This formula does not convey the intention, it merely means that there is some behavior in which, after providing an incorrect password, the door will be locked and a warning message will be displayed immediately afterwards. It does not give any guarantees with respect to the *"will be fairly performed"* part of the requirement, rather, it means that it "may be performed". Taking into account that this is a *response* pattern (under fairness assumption), the correct formulation should be:

$$[(\neg cmd(abb))^*.\ cmd(abb).\ pop\_calls(tuple(s\_locked,\ tuplenil(tuplenil(abc)))).\ (\neg warning\_message)]\ \langle true^*.\ warning\_message\rangle true$$

expressing that after an incorrect password is received, a locked door will be followed by a warning message.

Continuing the same line of research, in [97] the authors present a more involved case study, this time verifying an Erlang telecommunication system with a client-server architecture. It consists of a database server that maintains clients' data, and a number of functional servers (FS) that process the clients' requests. Each FS has a certain capacity defined as the maximum number of clients that can be connected simultaneously. Clients can communicate with any FS and perform typical operations such as calling. Upon disconnecting, the FS removes the client from its user list and releases the line. A number of properties are verified with CADP on an example system with two clients (m_1 and m_2) and a server (svr_1). For example, the property *when* m_1 *is connected to* svr_1 *and* m_2 *requests to* svr_1, svr_1 *will reply* m_2 *with busy* is formalized as:

$$\langle true^*.\ client\_info(m\_1,\ connected,\ svr\_1)^*.\ cmd(m\_2, connecting)^*.\ client\_info(m\_2,\ busy,\ svr\_1)\rangle true$$

With a similar culprit as in previous cases, this simply states that there are certain scenarios in which this behavior holds, it does not refute other possibly-incorrect ones. The correct formula can be obtained as a *response* pattern (an attempt of *connecting* should be followed by a *busy* reply) with an *after-Q* scope (after m_1 is connected to svr_1). In another experiment with two servers, the following property is checked:

$$\langle true^*.\ cmd(m\_2,\ connecting)^*.\ client\_info(m\_2,\ busy,\ svr\_1)^*.\ cmd(m\_2,\ connecting)^*.\ client\_info(m\_2,\ connected, svr\_2)\rangle true$$

stating that *when* m_2 *requests to connect to* svr_1 *and receives the reply of busy, it will*

request to connect to svr_2 and its request will be accepted by svr_2. The formula is wrong for the same reason as the previous case. The correct one should convey that, after receiving a *busy* signal, the client *will* (rather than *may*) request to connect to the second server, and this request will be accepted. This *response* behavior with an *after-Q* scope should be instantiated as:

$$[(\neg \mathit{client\_info}(m\_2,\ \mathit{busy},\ \mathit{svr}\_1))^*.\ \mathit{client\_info}(m\_2,\ \mathit{busy},\ \mathit{svr}\_1).\ \mathit{true}^*.\ \mathit{cmd}(m\_2,\ \mathit{connecting})]\ \ \mu X.\langle \mathit{true}\rangle \mathit{true} \wedge [\neg \mathit{client\_info}(m\_2,\ \mathit{connected}, \mathit{svr}\_2)]X$$

By putting under scrutiny this subset of existing research on formalizing properties, it becomes evident that shortcomings and flaws are almost unavoidable due to the complexity of the task. Observing these wrong formalizations, our conclusion is that typical mistakes include vacuously satisfied formulas, where an assumption (such as: two actions happen in immediate succession) is hidden in the property formulation. This is dangerous, as it may lead to the false conclusion that there are no problems in the underlying model, if the assumed property antecedent is false. Furthermore, using the modal operator *may* ($\langle \_ \rangle$) where *must* ($[\_]$) was intended, seems to be a recurring issue. This possibly stems from the more intuitive "linear view" on the notion of time, such as that used with LTL. In this view, time is treated as if each moment has a unique possible future, and formulas are interpreted over the behavior of each single program run. On the other hand, μ-calculus treats each moment in time as having multiple possible futures (branching time), thus interpreting the behavior over computation trees rather than linear structures. This is considered less intuitive. For example, the μ-calculus formula:

$$[\mathit{true}^*.\mathit{request}.(\neg \mathit{response})^*]\langle \mathit{true}^*.\mathit{response}\rangle \mathit{true}$$

is true in a computation tree if and only if in all its computations, **every** state in which a *request* action has been taken, branches to at least one computation in which *response* eventually happens. In contrast:

$$\langle \mathit{true}^*.\mathit{request}.(\neg \mathit{response})^*.\mathit{true}^*.\mathit{response}\rangle \mathit{true}$$

is already true if some state in which *request* happens, leads to at least one computation in which *response* eventually happens. In other words, it expresses that a process **can** do a *request* followed by *response*, rather than, as the former formula expresses, every *request* **will** be followed[7] by *response*. In most practical situations, the former behavior is desired. Although the goal of this survey was not to evaluate PASS directly, we conjecture that for most of the formulas the tool could have helped to avoid these common pitfalls.

[7] fairly, i.e., if not infinitely avoided.

5.7 Conclusions

To facilitate automated formal analysis and verification of concurrent systems, property specifications, in addition to models, need to be understandable and accessible to non-experts, while at the same time mathematically precise. For this reason we introduced PASS, a Property ASSistant that brings the process of correctly specifying functional properties closer to software engineers. Reusing ideas from [51], through a series of questions, the practitioner is guided to consider subtle aspects about a property which are often overlooked. Motivated by the wish to keep software engineers in the realm of their existing UML environment, rather than use an external helper tool, PASS was developed as an Eclipse plug-in, thus maintaining a strong relationship between the model elements and the property template ones. Our approach to specifying properties is based on the pattern system [67] and the μ-calculus mappings [134] provided by the CADP team, which we extended with over 150 pattern variations, deemed useful for model checking tools that work with the event-based modal μ-calculus.

Besides offering a natural language summary of the elicited property, a μ-calculus formula and a UML sequence diagram are provided, depicting the desired behavior. In addition, for a subclass of monitorable properties, PASS automatically generates monitors to be used for (potentially) more efficient property-driven runtime verification, also possible with the symbolic reachability tool LTSmin. We believe that automating the property specification process, while keeping practitioners in their familiar environment, should lead to more active adoption of methods for formal analysis of designs. We demonstrated the applicability of PASS for eliciting properties in real-world settings, by examining some misbehaving DIRAC subsystems. However, a thorough validation experiment with user tests should be conducted as part of a future work, to provide a more conclusive evidence on the potential effectiveness of PASS.

To reassess the usefulness of the pattern-based classification in event-based systems, we surveyed 25 published works that use μ-calculus to express system properties from different domains. In the context of the 178 properties encountered in the survey, our pattern extensions improve the standard patterns classification coverage by 10%. We observed a distribution of patterns which is consistent with surveys focusing on state-based systems. The results also indicate that subtle mistakes can easily be made in complex requirements formalization, potentially hiding behavioral problems in the models under verification.

Chapter 6

Conclusions

Experience is what you get when you didn't get what you wanted.

Randy Pausch

As software systems grow in size and complexity, it becomes more challenging to ensure that they behave correctly. This is especially true for distributed systems, where a multitude of components are running concurrently and rather autonomously (communicating and coordinating their actions at certain points in their execution), making it difficult to anticipate all the possible behaviors emerging in the system as a whole. Certain design errors, such as deadlocks and race-conditions, can often go unnoticed when testing is the only form of verification employed in the common software engineering life-cycle. Even when these problems have manifested and are detected in a running software, revealing the root cause and reproducing the behavior can be time consuming (and even impossible), given the lack of control the engineer has over the execution of the concurrent components, as well as the number of possible scenarios that could have produced the problem. This is especially pronounced for large-scale distributed systems such as the Worldwide LHC Computing Grid (WLCG [5]).

Formal verification methods offer more rigorous means of determining whether a system satisfies certain behavioral requirements, and one highly effective verification technique is model checking. Model checking is a mathematically-rooted algorithmic procedure, nowadays automated by many actively-maintained and mature tools. There are plenty of case studies of applying model checking in academia [30; 16; 180; 104; 145], and to a certain extent in industrial labs [18; 73; 13], revealing critical bugs undetected before. The drawback, however, is the necessity to be proficient in formal language notations such as process algebras and temporal logics, essential for describing a model of the system and the behavioral requirements to be verified by these tools. In effect, the application of model checking seems to be mostly focused on hardware, communication protocols, device drivers, and safety-critical components, which are typically well structured, deterministic, or with a

relatively small code-base. There are some success stories of applying model checking in other domains, and at a larger-scale [41; 193; 111], but such practice is far less frequent, to the best of our knowledge.

In the remainder of this chapter we summarize the thesis results, followed by a discussion on the assumptions and limitations, and finally we suggest possible future work directions.

6.1 Thesis summary

Inspired by the complexity of the DIRAC software (partly elaborated in Chapter 2), the main question addressed in this thesis is *how to integrate model checking into the common software development cycle of realistic-scale, distributed, data-driven software, by automating the aspects that require formal methods expertise*.

Prior to answering this question, we first had to select a formalism suitable and powerful enough for modeling and addressing design errors that are common for such software. In Chapter 3 we studied the feasibility of using the mCRL2 language for this purpose, by systematically abstracting the source code and modeling the behavior of two DIRAC subsystems. The case studies indicated that mCRL2 has the necessary concurrency, data abstraction and manipulation mechanisms to faithfully model the subsystems. Furthermore, the mCRL2 toolset offered the analysis tools needed to discover critical race conditions and livelocks, which were confirmed to occur in DIRAC. By simulating, visualizing, and model checking the resulting models with the mCRL2 toolset, we were also able to gain a better insight into the system behavior, and replaying the counter-example traces helped in localizing the detected problems. Taking into account that each release of DIRAC undergoes testing and certification, before running in production, the advantages of using model checking become apparent. Although some of the faulty behaviors already manifested in production before the subsystems were formally modeled, they were not localized within the timespan of the modeling & verification we performed.

Manually constructing a formal model based on analysis of a software implementation can be time consuming. In addition to the greater risk of making modeling mistakes, in order to be useful, the formal model must be kept up to date with the continuous software updates. Software implementations contain too many language-specific details to serve as footprints for deriving formal models. It is a common practice in software engineering to provide higher-level visual designs for communicating and validating the requirements, before the implementation and testing takes place. UML is generally accepted as a visual modeling language for this purpose, supported by several widely adopted CASE tools. However, due to the lack of a formal semantics definition, it is not directly usable for formal verification methods such as model checking. There are numerous works on formalizing different structural and behavioral UML diagram types in the literature. In Chapter 4

we presented a transformation approach for automatically deriving mCRL2 formal models, based on UML sequence and activity diagrams. We discussed the semantic choices we made with respect to some ambiguities in the (official) semi-formal semantics, and compared our approach with the existing ones, in the context of a well-known classification. The transformation preserves the object-oriented structure of the system, facilitating a straightforward round-trip approach in which the verification counter-examples can also be visually presented as sequence diagrams. To provide some empirical evidence and confidence in the correctness of the translation, we applied the tool-supported approach on DIRAC's new workload system functionality. We discovered the root cause of a logical flaw leading to no-progress, which was observed earlier in the testing phase of this functionality.

To be amenable to automated model checking, behavioral properties must be expressed as formulas in temporal logic. The mCRL2 toolset requires the use of the modal μ-calculus for this; a very expressive logic, but not very intuitive nor accessible, compared to simpler logics like LTL and CTL. Behavioral requirements for software are typically expressed in natural language, and as such can be subjects to different interpretations. In the context of the main research question, to bring the process of correctly eliciting behavioral properties closer to software engineers, in Chapter 5 we introduced a property assistant tool - PASS, as part of a UML-based front-end to the mCRL2 toolset. Based on a well-known property pattern classification from Dwyer et al. [67], which we extended with new pattern variations for the event-based modal μ-calculus, the tool provides assistance to non-experts in eliciting properties which capture the required behavior precisely and unambiguously. Besides a μ-calculus formula, the tool outputs a natural language summary, and a UML sequence diagram depicting the property. In addition, for a subclass of monitorable properties, PASS automatically generates monitors to be used for (potentially) more efficient property-driven runtime verification. We demonstrated the usage of PASS on DIRAC's behavior.

Dwyer et al.'s property pattern classification is based on a large literature survey of how specification formalisms are used in practise. Since we did not encounter any μ-calculus formulas in their collection repository, we were curious whether its usage differs significantly from other formalisms. We surveyed 25 published works that use μ-calculus to express system properties from different domains. From the 178 properties in the survey, our pattern extensions improve the standard patterns classification coverage by 10%. The modal μ-calculus, while considered less intuitive compared to the more commonly used ones (like LTL and CTL), is able to capture certain property patterns which either LTL or CTL cannot. We observed a distribution of patterns which is rather consistent with similar surveys focusing on other formalisms. The results also indicate that subtle mistakes can easily be made when constructing temporal logic formulas manually.

6.2 Limitations and Future Directions

To claim that the work in this thesis has fully succeeded in the quest for a definite closure of the gap between software engineering and model checking, would be disingenuous. One will almost always have had more research ideas, than one could manage to carry out. In the following, we briefly re-iterate assumptions we made, discuss some of the limitations, and suggest possible future work directions.

Relying on UML designs as a source for building formal behavioral models imposes certain limitations. Large-scale software nowadays almost unavoidably re-uses existing software components, usually as external libraries. DIRAC is no exception to this. The usage of external libraries in UML is reflected only in structural diagrams (Component diagrams, in particular). This means that we need to make certain assumptions when abstracting the behavior at the boundaries of the modeled system. This assumed behavior at the boundaries can be included in sequence diagrams by mimicking the functionality provided by the external libraries, and wrong assumptions can lead to incorrect models, obviously. The assumptions must be carefully reviewed when a property violation occurs, which involves the usage of such an external library.

Incomplete models are common in the early phases of the software engineering life-cycle, when they are often defined at a higher level of abstraction, with refinements taking place later. The model transformation approach described in Chapter 4 assumes that all behavior about the system is present in the UML design. This requirement can be seen as too strict, depending on the precise meaning of "incomplete". For instance, while some behavior (like method calls, or entire components) may be left unspecified, a *loop* combined fragment with a missing Boolean condition to specify the loop termination, is not permitted. Process algebra models, like the ones written in mCRL2, are closed, meaning that the behavior which is not modeled, is assumed not to happen. Leaving out behavior unspecified is generally permitted, but then there is a risk that design errors are hidden in the abstraction. Underspecified modeled behavior is in most cases not permitted, although there are certain defaults which the transformation assumes, in absence of information (e.g., if the multiplicity of a method argument is not specified, it is assumed to be 1). Verification of partial and incomplete models is still an open research topic. For example, multi-valued model checking [98] could be useful for analyzing models that contain uncertainty or inconsistency. In [33], a method called *model-based integration and testing* was developed, which allows to integrate formal (executable) models of system components that are not yet physically realized, with available realizations of other components. Models should exist before their implementations. Coupling them with the already-available component realizations via an appropriate infrastructure enables earlier detection and prevention of problems, compared

to real system testing and integration. The potential of the proposed method is demonstrated by its application to the development of a part of a realistic industrial system, namely the ASML wafer scanner.

The assumption that UML models, and in particular sequence diagrams, are always available for a software, may also not hold. Such is the case with DIRAC, for example. Reverse engineering sequence diagrams has been the focus of research for a while now [35; 22; 169; 94]. However, tool-supported approaches we came across are scarce, restricted to Java, and based on collecting traces of program execution. The trace data is then translated into sequence diagrams, using higher-level analysis to synthesize loops, alternative and parallel operator fragments. Recovering sequence diagrams from program execution traces has practical limitations. While it may be more useful for web applications, large-scale software which is highly data-driven will almost certainly result in incomplete, and even incorrect models. Nevertheless, they could serve as a starting point. It would be interesting to further investigate the feasibility of using such an approach on a software like DIRAC, on the longer run. Perhaps a tighter integration of static (code) and dynamic (execution) analysis will deliver better results in future research.

Given that the approach in Chapter 4 transforms each UML class method into a separate mCRL2 process definition, and the transformation preserves most of the object-oriented information, process proliferation is likely to happen for larger UML models, leading to a state space explosion. In the Executor Framework case study of Section 4.4 there are as many as 50 processes, with a generated model of over 2k lines of mCRL2 code, and already generating the entire state space becomes problematic at this scale. Using the generated monitor for a given property, when possible, in order to turn the model checking problem into a reachability one, is one way to tackle the problem. The mCRL2 and LTSmin toolsets have multiple different model exploration and minimization strategies which can be employed, however, there is no "winning" combination which works with the same efficiency for different models. This still requires a certain level of ingenuity, if options beyond the default ones are to be used. These toolsets constantly evolve and improve and are able to handle large models. Nevertheless, state space explosion can be a serious impediment. Our approach can be improved with some optimizations. For example, nested private method calls within a class can be "flattened" into a single mCRL2 process; similarly, if a particular sequence of method calls always happens in the same order, they can be "merged" into a single mCRL2 process. At the time of writing, the multi-core [123] LTSmin backend was the only available tool exploiting parallel algorithms for model checking mCRL2 models. However, applying it to the Executor Framework and the SMS did not bring any obvious benefits compared to sequential exploration. There could be various reasons for this. The amount of parallelism in our models may not be sufficient for the multi-core algorithms to really

scale. The size of the data domain in the models is probably the dominant resource consumer, incurring significant memory overheads, which cannot be avoided in the multi-core setting either.

In general, if a property violation is found, a counter-example trace is visualized as a sequence diagram. However, a practical limitation of the counter-example visualization surfaces when checking liveness properties using the symbolic PBES tools from mCRL2. As already stated earlier, a violation of liveness properties can only be detected by an infinite execution trace, and visualizing the "cryptic" trees of PBES variables instantiations that the tools produce in such case, is not tackled in this thesis. These tools do not (yet) provide feedback in terms of some counter-example in case the property fails to hold. A potential solution to this problem has been recently addressed in [56].

The criticism that a more systematic evaluation of the PASS tool presented in Chapter 5 is needed, is clearly pertinent. The main reason is a practical one of timing constraints: performing such evaluation would require setting up controlled experiments and involving more users over a longer time span.

We have shown that the approach taken in this thesis can be useful, but this work is far from a complete push-button solution to the problem of making model checking a common and largely automated software engineering practise. We hope, however, that the results presented here will inspire further research into the field, until the gap between these two disciplines is finally closed.

As one possible direction for future work, it would be interesting to explore beyond discovery of behavioral problems, in particular in the area of performance assessment of models. The SPT/MARTE [93] UML profiles provide support for annotating sequence diagrams with quantitative information such as time, throughput, and resource usage. Although this type of analysis is a primary concern for the domain of real-time and embedded systems, valuable insight can also be gained about communication costs, delays, and various bottlenecks in distributed systems. Using these profiles in the context of formal verification has been a popular area of research [175; 150; 43; 199] for some time. Adding this kind of information to traditional process algebra models means getting in the domain of stochastic process algebra. Random variables with exponential distribution are typically associated with the designated events, to express system transitions with duration, which could be estimates of processing time or communication overhead. We do not necessarily have to use another formalism for enhancing the models with stochastic information. The CADP toolset for analysis of stochastic models is well integrated with mCRL2 for this purpose. CADP can explore and manipulate mCRL2 specifications via an interface for a common LTS format representation [80] which both tools understand. However, in such case, the state space explosion problem will obviously become even more pronounced.

Samenvatting (Summary in Dutch)

Doordat software voortdurend groeit in grootte en complexiteit is het een steeds grotere uitdaging om er voor te zorgen dat het zich correct gedraagt. Dit geldt vooral voor gedistribueerde systemen waar een veelheid aan componenten gelijktijdig draaien. Hierdoor is het moeilijk om al het mogelijke gedrag van het systeem als geheel te anticiperen. Sommige ontwerpfouten, zoals impasses en race-condities, blijven vaak onopgemerkt als het testen van software de enige vorm van controle is tijdens de ontwerpcyclus van software. Zelfs als fouten ontdekt worden tijdens het draaien van software dan is het vaak tijdrovend (of zelfs onmogelijk) om de uiteindelijke oorzaak te achterhalen doordat de programmeur nauwelijks invloed heeft op de uitvoering van onderdelen die gelijktijdig draaien en door de hoeveelheid mogelijke scenario's die het probleem veroorzaakt kunnen hebben. Dit geldt vooral voor grootschalig gedistribueerde systemen zoals het Worldwide Large Hadron Collider Computing Grid (Wereldwijd Grote Hadronen-Botser Computernetwerk).

Formele verificatie methoden bieden grondiger manieren om te bepalen of een systeem voldoet aan bepaalde vereisten met betrekking tot gedrag. Een erg effectieve verificatie techniek is modelchecken. Modelchecken is een op wiskunde gebaseerd algoritmische procedure die tegenwoordig automatisch gedaan wordt door vele actief onderhouden en uitontwikkelde gereedschappen. Een nadeel is echter dat het nodig is om bedreven te zijn in formele-taalnotaties zoals procesalgebra en temporele logica die onontbeerlijk zijn voor zowel de beschrijving van het model van het systeem en de eisen aan het gedrag die door deze gereedschappen getoetst moeten worden.

In dit proefschrift beantwoorden we de vraag hoe modelchecken geïntegreerd kan worden in de gebruikelijke ontwikkelingscyclus van distribueerde, datagedreven software met een realistische grootte, met het automatiseren van die aspecten die gespecialiseerde kennis van formele methoden vereisen. Voor dit doel kozen we een formalisme dat krachtig genoeg is om ontwerpfouten die regelmatig voorkomen in dergelijke software's te modelleren en aan te pakken. De aanleiding is DIRAC, een raamwerk voor gedistribueerde "grid" software dat wordt ontwikkeld en gebruikt

door de fysicagemeenschap van de Europese organisatie voor deeltjesfysica (CERN). We onderzoeken of het mogelijk is om de mCRL2 taal te gebruiken door van twee deelsystemen van DIRAC systematisch de broncode te abstraheren en het gedrag te modelleren. De studies geven aan dat mCRL2 de nodige mechanismen heeft om van de deelsystemen de gelijktijdigheid, data-abstractie en manipulatie getrouw te modelleren. Door simulatie, visualisatie en toetsing van de resulterende modellen met de mCRL2 gereedschapskist werd het voor ons mogelijk om een beter inzicht te krijgen in het gedrag van het systeem. Het afspelen van tegenvoorbeelden hielp in het lokaliseren van de opgemerkte problemen. Hoewel een deel van het foutieve gedrag al in de praktijk naar voren was gekomen voordat de deelsystemen formeel gemodelleerd waren, was het nog niet gelokaliseerd binnen de tijdspanne waarin we de modellering en verificatie uitgevoerd hebben.

Het met de hand construeren van een formeel model voor een concrete software kan tijdrovend zijn. Daarbij is er het risico om fouten te maken. Bovendien, moet het model bijgewerkt blijven bij voortdurende veranderingen aan software, om nuttig te zijn. Programma's bevatten te veel taalspecifieke details om als uitgangspunt te dienen voor het afleiden van modellen. Bij software engineering worden hoogniveau visuele ontwerpen gemaakt voor overleg over en het valideren van vereisten voordat de uitwerking en het testen plaats vinden. Universal Modeling Language (UML, universele modelleringstaal) wordt algemeen geaccepteerd als een visuele modelleertaal voor dit doel. We presenteren een aanpak voor het automatisch afleiden van een mCRL2 formeel model door de transformatie van UML sequentie- en activiteitsdiagrammen. We bespreken de keuzes op het gebied van semantiek die we gemaakt hebben waar het gaat om ambiguïteiten in officiële halfformele UML-semantiek en we vergelijken onze aanpak met een al bestaande aanpak binnen het kader van een bekende classificatie. De transformatie bewaart de object georiëneerde structuur van het systeem en maakt het eenvoudig om tegenvoorbeelden van verificatie visueel te presenteren als sequentiediagrammen. Om wat praktijk ervaring en vertrouwen te wekken in de juistheid van de transformaties passen we de gereedschapondersteunde aanpak toe op het nieuwe werkbelastingsysteem van DIRAC. We ontdekken de uiteindelijke oorzaak van een logische fout die er voor zorgt dat er geen vooruitgang meer geboekt wordt. Een fout die eerder tijdens testfase van deze functie waargenomen is.

In modelchecken moeten eigenschappen van gedrag uitgedrukt worden als formules in temporele logica. De mCRL2 gereedschapskist vereist het gebruik van modale μ-calculus voor dit doel; een erg expressieve logica, maar niet erg intuïtief of toegankelijk. Vereisten voor gedrag van software worden normaal gesproken uitgedrukt in natuurlijk taal en zijn als zodanig op meerdere manieren uit te leggen. In het kader van de centrale onderzoeksvraag, het correct uitdrukken van gedragseigenschappen dichter bij programmeurs te brengen, voegen we een eigenscha-

passistent met de naam PASS toe aan de op UML gebaseerde frontend van de mCRL2 gereedschappen. Dit gereedschap assisteert leken in het naar voren halen van eigenschappen die het vereiste gedrag precies en niet-ambigu uitdrukken. Het is gebaseerd op een bekende classificatie van patronen van eigenschappen die we uitgebreid hebben met nieuwe patroonvariaties voor de event-based modale μ-calculus. Naast de μ-calculus formules biedt het gereedschap een samenvatting in natuurlijke taal en een UML sequentiediagram dat de eigenschap afbeeldt. Verder genereert PASS automatisch voor een deel van de eigenschappen observatiestructuren die (potentieel) gebruikt kunnen worden voor de efficiëntere eigenschapgedreven verificatie tijdens de uitvoering. We demonstreren het gebruik van PASS met betrekking tot het gedrag van DIRAC.

De classificatie van eigenschappatronen is gebaseerd op een groot literatuuronderzoek naar het gebruik van specificatieformalismen in de praktijk. Aangezien we geen μ-calculus formules aantroffen in de verzameling waren we benieuwd of het gebruik significant afwijkt van dat van andere formalismen. We hebben 25 gepubliceerde werken in kaart gebracht die μ-calculus gebruiken om eigenschappen uit te drukken van systemen uit verschillende domeinen. Ten opzichte van de 178 eigenschappen uit het literatuuronderzoek bieden onze extra patronen een 10% extra dekking van de geclassificeerde patronen. De modale μ-calculus, die als minder intuïtief beschouwd wordt dan meer algemene logica's (zoals LTL - Lineaire Temporele Logica, en CTL - Computationele Boom Logica), is in staat sommige patronen van eigenschappen te omvatten waartoe LTL en CTL afzonderlijk niet in staat zijn. We hebben een verdeling van de patronen waargenomen die behoorlijk consistent is met vergelijkbare onderzoeken naar andere formalismen. De resultaten geven ook aan dat het eenvoudig is om subtiele fouten te maken bij het handmatig samenstellen van temporele logicaformules.

Als een mogelijke toekomstige richting zou het interessant zijn om dit werk uit te breiden buiten het ontdekken van gedragsproblemen, in het bijzonder op het gebied van het inschatten van prestaties aan de hand van UML modellen. Hoewel dit type analyse van primair belang is voor "real-time"- en embedded systemen kunnen ook waardevolle inzichten verkregen worden over de communicatiekosten, vertragingen en verschillende flessenhalzen in gedistribueerde systemen.

Appendix A

Appendix

A.1 Definitions

Structural Operational Semantics (SOS) is a means to define the operational semantics, i.e., the actions that the process, described by a piece of syntax, can perform. The SOS for the mCRL2 process algebra is represented by a Labeled Transition System (LTS). The behavior of the LTS is defined in terms of the behavior of its parts inductively, defining the valid transitions of a piece of syntax, in terms of the transitions of its components (structure).

In this section we formally define the syntactic elements of mCRL2 and the associated semantics in SOS. We also provide the formal semantics of modal μ-calculus formulas. By providing such mapping of the syntactic constructs to a mathematical domain, we have the necessary grounds to prove the monitor construction correctness. We restrict ourselves to the untimed fragment of the mCRL2 language, as used throughout the thesis. Readers interested in the full syntax and semantics of mCRL2 (including the timed fragment), can refer to [84].

For the following definitions, we assume that processes are interpreted in the presence of a data model ($\mathcal{D}$-model), providing an interpretation (or valuation) σ for assigning values to variables. The valuation maps all variables $x : D$ (where D is a sort) into a corresponding semantical domain M_D, i.e., $\sigma(x) \in M_D$. In the presence of a $\mathcal{D}$-model, we assume that for each element $e \in M_D$, the syntactic notation for e is t_e. In this $\mathcal{D}$-model, $M_{\mathbb{B}}$ is a set with two elements of Boolean sort, denoted by **true** and **false** such that **true**≠**false** holds. We write $\sigma[d/x]$ for a valuation with function update that maps all variables according to σ except for x. This variable is mapped to the value d. We write $\sigma[d_i/x_i]_{1\leq i\leq n}$ for the valuation that maps all variables according to σ except that it maps each variable x_i to d_i. The interpretation function extends trivially to data expressions. For example, $\sigma(true) =$

true and $\sigma(false) = \textbf{false}$ for every valuation σ, i.e., the elements represent the truth values **true** and **false**.

In Chapter 3 we introduced the syntax of the mCRL2 language and we sketched how it is used to describe process behavior. We now re-iterate the syntax and give more formal definitions of action declarations, syntactic multi-actions, and process expressions.

Definition A.1.1 (**Action declaration**). An *action declaration* is an expression of the form $a : D_1 \times \cdots \times D_n$ where $n \geq 0$, and for all $0 \leq i \leq n, D_i$ are sorts.
An action with action label a and data expressions $d_1, \ldots d_n$ is denoted as $a(d_1, \ldots d_n)$.
Note that data parameters in the syntactic (multi-)actions are data expressions and that the data parameters in the semantic (multi-)actions are values. The set of action labels is shared among the syntactic and semantic (multi-)actions.

Definition A.1.2 (**Syntactic multi-actions**). Let $\mathcal{A}$ be a set of action labels. A *syntactic multi-action* is a collection of actions that are specified to occur at the same time instant. They have the following BNF:

$$\alpha ::= \tau \mid a(d_1, \ldots, a_n) \mid \alpha|\beta$$

Here, τ represents the empty multi-action, the term $a \in \mathcal{A}$ is an action label and $(d_1, \ldots d_n)$ is a vector of data expressions of sorts $(D_1, \ldots, D_n)$ respectively, while $\alpha|\beta$ are syntactic multi-actions.

Definition A.1.3 (**Process expression**). *Process expressions* are expressions with the following BNF syntax:

$$p ::= \alpha \mid p + p \mid p \cdot p \mid \delta \mid c \to p \mid c \to p \diamond p \mid \sum_{v:D} p \mid p \| p \mid p|p \mid$$

$$\Gamma_C(p) \mid \nabla_V(p) \mid \partial_B(p) \mid \rho_R(p) \mid \tau_I(p) \mid$$

$$X \mid X() \mid X(d_1, \ldots, d_n) \mid X(v_1 = d_1, \ldots, v_n = d_n)$$

Here, α is a multi-action, c is a Boolean data expression, $v, v_1, \ldots, v_n$ are variables, D is a sort, C is a set of communications, V is a set of multi-action labels, I and B are sets of action labels, R is a set of renamings, X is a process name and $d_1, \ldots, d_n$ are data expressions.

Definition A.1.4 (**Process equation**). A *process equation* is an expression of the form $X(d_1 : D_1, \ldots, d_n : D_n) = p$ where $n \geq 0$, and d_i are variables of sorts D_i, and p is a process expression.

Definition A.1.5 (**Process specification**). A *process specification* is a five tuple $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ where:

- $\mathcal{D}$ is a data specification
- AD is a set of action declarations
- PE is a set of process equations
- p is a closed process expression (i.e., one without free data variables), called the *initial* process
- $\mathcal{X}$ is a set of global variables.

For simplicity, we also omit the formal definitions (available in [84]) on well-typed process specifications, and assume that all their underlying components are well-typed as well, i.e., the syntactic constructs defined here guarantee that they have semantics. The semantics of a process is defined using operational (inference) rules, which use several notations on semantical multi-actions. We define these below.

Definition A.1.6 (Semantics of multi-actions). In the presence of a $\mathcal{D}$-model, we inductively define the semantical interpretation $[\![\cdot]\!]^\sigma$ of a multi-action α for any data valuation σ as follows:

- $[\![\tau]\!]^\sigma = \tau$
- $[\![a(t_1, \ldots, t_n)]\!]^\sigma = a([\![t_1]\!]^\sigma, \cdots, [\![t_n]\!]^\sigma)$
- $[\![\alpha|\beta]\!]^\sigma = [\![\alpha]\!]^\sigma | [\![\beta]\!]^\sigma$

In other words, a semantical multi-action is the interpretation of a syntactic multi-action, in which the vector of data expressions $(t_1, \ldots, t_n)$ are interpreted under a data valuation σ. Here $|$ at the right-hand side is a new operator on semantical multi-actions. We also use τ for the semantical empty multi-action, but use ω and $\overline{\omega}$ for semantical multi-actions in general.

Definition A.1.7. Let ω be a semantical multi-action. We use the following notations for the SOS rules defined in Table A.5:

- $\omega_{\{\}}$ is the set of all actions occurring in ω. In particular $a_{\{\}} = \{a\}$, $(\alpha|\beta)_{\{\}} = \alpha_{\{\}} \cup \beta_{\{\}}$ and $\tau_{\{\}} = \emptyset$.
- Let R be a set of renamings. $R \bullet \omega$ is the semantical multi-action where the labels have been renamed according to R. More precisely: $R \bullet a(d_1, \ldots, d_n) = b(d_1, \ldots, d_n)$ if $a \to b \in R$. Otherwise the result is just $a(d_1, \ldots, d_n)$. One particular renaming operator we use is $\eta_I(\omega)$, which renames all action labels in I to the (visible) internal action int. Furthermore, we use $\theta_I(\omega)$ which removes all actions with labels that occur in I.

- We use $\underline{\omega}$ for the multi-action ω where all data parameters are removed. In particular, $\underline{a(t_1, \ldots, t_n)} = a$.
- Communication is defined using γ_C. It says that a communication within different actions in a multi-action takes place, exactly when this communication occurs in C and the arguments of these actions have the same data arguments. By defining $\gamma_{C_1 \cup C_2}(\omega) = \gamma_{C_1}(\gamma_{C_2}(\omega))$ we can apply each renaming on its own:
  $\gamma_{\{a_1|\ldots|a_n \to b\}}(\omega) =$

$$\begin{cases} b(\overrightarrow{d})|\gamma_{\{a_1|\ldots|a_n \to b\}}(\overline{\omega}) & \text{if actions } a_i(\overrightarrow{d}) \text{ occur in } \omega \text{ for all } 1 \leq i \leq n \\ \alpha & \text{otherwise.} \end{cases}$$

  where $\overline{\omega} = \omega \setminus (a_1(\overrightarrow{d})|\ldots|a_n(\overrightarrow{d}))$, i.e., the multi-action ω from which the actions $a_i(\overrightarrow{d})$ are removed.

Definition A.1.8 (Semantics of a process). Let $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ be a process specification. We use a substitution ξ mapping the variables in $\mathcal{X}$ to closed data expressions. For the definition of the substitution function ξ as well as closed data and process expressions, please refer to [84].
We define the semantics of PS as a labeled transition system $A = (S, Act, \longrightarrow, s_0, T)$ as follows:

- The states S are closed process expressions p' with syntax given in Def. A.1.3. In addition, there is one special termination state, denoted by $\surd$.
- The transitions are inductively defined by the operational rules in Tables A.1, A.2, A.3, A.4, A.5, and A.6. The transition relation is denoted by $p' \xrightarrow{\omega} p''$ or $p' \xrightarrow{\omega} \surd$.
- The set Act of labels contains the semantical multi-actions.
- The initial state s_0 is $\xi(p)$.
- The set of successfully terminating states T contains only $\surd$,i.e., $T = \{\surd\}$.

Given a labeled transition system that forms the semantics of a process expression, we can define in which states a modal μ-calculus formula holds. To do so, we first define the semantics of an action formula, a regular formula, and finally, a modal μ-calculus formula.

Definition A.1.9 (**Semantics of action formulas**). Let $A = (S, Act, \longrightarrow, s_0, T)$ be a labeled transition system where Act consists of all semantical (multi-)actions. For the syntax of action formulas α, given by the following grammar:

$$\alpha_1, \alpha_2 ::= b \mid \alpha(e) \mid \neg\alpha \mid \alpha_1 \cap \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \exists d{:}D.\alpha$$

we define the semantics, denoted $[\![\alpha]\!]^\sigma$, where σ is a valuation, as a set of semantical (multi-)actions given inductively by:

$$\begin{aligned}
[\![b]\!]^\sigma &= \begin{cases} Act & \text{if } \sigma(b) = \textbf{true}, \\ \emptyset & \text{otherwise.} \end{cases} \\
[\![\alpha(e)]\!]^\sigma &= \{\alpha([\![e]\!]^\sigma)\} \\
[\![\neg(\alpha)]\!]^\sigma &= Act \setminus [\![\alpha]\!]^\sigma \\
[\![\alpha_1 \cap \alpha_2]\!]^\sigma &= [\![\alpha_1]\!]^\sigma \cap [\![\alpha_2]\!]^\sigma \\
[\![\alpha_1 \cup \alpha_2]\!]^\sigma &= [\![\alpha_1]\!]^\sigma \cup [\![\alpha_2]\!]^\sigma \\
[\![\exists d{:}D.\alpha]\!]^\sigma &= \bigcup_{x \in M_D} [\![\alpha]\!]^{\sigma[x/d]}
\end{aligned}$$

Here $[\![e]\!]^\sigma$ denotes the semantics of the data expression e in an environment σ. Note that the occurrence of $[\![b]\!]^\sigma$ on the left-hand side denotes the action formula semantics (i.e., a set of actions), while the right-hand side $\sigma(b)$ denotes the Boolean expression's semantics, i.e., **true** or **false**.

We further define the semantics of regular formulas. A regular formula R allows the use of regular expressions over action formulas.

Definition A.1.10 (**Semantics of regular formulas**). Let $A = (S, Act, \longrightarrow, s_0, T)$ be a labeled transition system where Act consists of all semantical (multi-)actions. For the syntax of regular formulas R, given by the following grammar:

$$R ::= \alpha \mid R \cdot R \mid R + R$$

we inductively define the semantics, denoted $\|R\|^\sigma \subseteq Act^*$, where σ is a valuation, as a word over the alphabet Act:

$$\begin{aligned}
\|\alpha\|^\sigma &= [\![\alpha]\!]^\sigma \\
\|R_1 \cdot R_2\|^\sigma &= \|R_1\|^\sigma \circ \|R_2\|^\sigma
\end{aligned}$$

$$\|R_1 + R_2\|^\sigma = \|R_1\|^\sigma \cup \|R_2\|^\sigma$$

In the semantic interpretation $\|R\|$, the operators $\circ$ and $\cup$ denote the concatenation and union operations on sets. The α regular formula characterizes single-action words a such that $a \in [\![\alpha]\!]$.

Definition A.1.11 (Semantics of modal μ-calculus formulas).

Let $A = (S, Act, \longrightarrow, s_0, T)$ be a labeled transition system where Act consists of all semantical (multi-)actions. We consider only a fragment of the modal μ-calculus formulas syntax necessary for the monitor construction proof, given by the following grammar:

$$\phi ::= b \mid \neg\phi \mid \phi \wedge \phi \mid \forall d{:}D.\phi \mid [R]\phi$$

We define the semantics of a modal μ-calculus formula ϕ, denoted $[\![\phi]\!]^{\sigma,\rho}$, where σ is a valuation, and ρ is a logical variable valuation, as a set of states of the labeled transition system where ϕ is valid, by:

$$\begin{aligned}
[\![b]\!]^{\sigma,\rho} &= \begin{cases} S & \text{if } \sigma(b) = \textbf{true}, \\ \emptyset & \text{otherwise.} \end{cases} \\
[\![\neg\phi]\!]^{\sigma,\rho} &= S \setminus [\![\phi]\!]^{\sigma,\rho} \\
[\![\phi_1 \wedge \phi_2]\!]^{\sigma,\rho} &= [\![\phi_1]\!]^{\sigma,\rho} \cap [\![\phi_2]\!]^{\sigma,\rho} \\
[\![\forall d{:}D.\phi]\!]^{\sigma,\rho} &= \bigcap_{x \in M_D} [\![\phi]\!]^{\sigma[x/d],\rho} \\
[\![[R]\phi]\!]^{\sigma,\rho} &= \{s \in S \mid \forall s' \in S, \alpha \in \|R\|^\sigma : s \xrightarrow{\alpha} s' \wedge \alpha \in \|R\|^\sigma \Rightarrow s' \in [\![\phi]\!]^{\sigma,\rho}\}
\end{aligned}$$

Here $[\![e]\!]^\sigma$ denotes the semantics of the data expression e in an environment σ. We say that a formula ϕ holds in A iff $s_0 \in [\![\phi]\!]^{\sigma,\rho}$ for any σ, ρ and $[\![\cdot]\!]$. Note that we omit the definition of the semantics of the fixpoint operators μ and ν, and the diamond modality $\langle\_\rangle$, as they are not necessary for proving the monitor correctness.

A.2 Proof of the Monitor Construction Correctness

Lemma 1. Let η be an arbitrary valuation function for data variables, and let t_η be a substitution defined as $t_\eta(v) = t_d$ iff $\eta(v) = d$ for $d \in M_D$. For all action formulas α that do not contain variable v, and all lists l, we have that the action $a(d) \in [\![\alpha]\!]^\eta$ iff $[\![\mathsf{Cond}_l(a(v), \alpha)\ t_\eta[v := t_d]]\!]$ holds.

Proof. We use structural induction on the structure of the action formula α. Let η be arbitrary.

- Case $\alpha \equiv b$. Then, by Def. A.1.9: **Semantics of action formulas** $[\![b]\!]^\eta = Act$ if $\eta(b)$ holds and $[\![b]\!]^\eta = \emptyset$ otherwise. We distinguish two cases.

  1. Case $\eta(b)$ holds. Then (Def. A.1.9: **Semantics of action formulas**) $[\![b]\!]^\eta = Act$. Clearly, for all values $d \in M_D$ for which $[\![\mathsf{Cond}_l(a(d), b)\ t_\eta[v := t_d]]\!]$ holds, we have $a(d) \in Act$. For the reverse, let $a(d) \in Act$. We show that then $[\![\mathsf{Cond}_l(a(v), b)\ t_\eta[v := t_d]]\!]$ holds.
     We have $[\![\mathsf{Cond}_l(a(v), b)\ t_\eta[v := t_d]]\!]$ = (monitor definition) = $[\![b\ t_\eta[v := t_d]]\!] = [\![b]\!]^\eta = \eta(b)$, which holds, by assumption.
  2. Case $\eta(b)$ does not hold. Then (Def. A.1.9: **Semantics of action formulas**) $[\![b]\!]^\eta = \emptyset$. We must show that then $[\![\mathsf{Cond}_l(a(v), b)\ t_\eta[v := t_d]]\!]$ does not hold for all $a(d)$. Let $a(d) \in Act$ be arbitrary.
     We have $[\![\mathsf{Cond}_l(a(v), b)\ t_\eta[v := t_d]]\!]$ = (monitor definition) =$[\![b\ t_\eta[v := t_d]]\!] = [\![b]\!]^\eta = \eta(b)$, which does not hold, by assumption.

- Case $\alpha \equiv a'(e)$. Then, by Def. A.1.9: **Semantics of action formulas** $[\![a'(e)]\!]^\eta = \{a'([\![e]\!]^\eta)\} = \{a'(\eta(e))\}$. Let $a(d)$ be arbitrary. We distinguish two cases.

  1. Case $a = a'$. Then we have $[\![\mathsf{Cond}_l(a(v), a'(e))\ t_\eta[v := t_d]]\!]$ = (monitor definition) $[\![v = e\ t_\eta[v := t_d]]\!] = [\![true\ t_\eta[v := t_d]]\!]$ iff $d = \eta(e)$, which holds iff $a(d) \in \{a'(\eta(e))\}$.
  2. Case $a \neq a'$. Then we have $[\![\mathsf{Cond}_l(a(v), a'(e))\ t_\eta[v := t_d]]\!]$ = (monitor definition) = $[\![false\ t_\eta[v := t_d]]\!]$ which does not hold. But we also do not have $a(d) \in \{a'(\eta(e))\}$ in that case.

  Induction hypothesis: For any valuation η, it holds that $a(d) \in [\![\alpha]\!]^\eta$ iff $[\![\mathsf{Cond}_l(a(v), \alpha)\ t_\eta[v := t_d]]\!]$ holds.

- Case $\alpha \equiv \neg\alpha_1$, and our statement holds by induction for α_1. By definition, we have $a(d) \in [\![\alpha]\!]^\eta$ iff $a(d) \in [\![\neg\alpha_1]\!]^\eta$ iff (Def. A.1.9: **Semantics of action formulas**) $a(d) \in Act \setminus [\![\alpha_1]\!]^\eta$ iff (set theory) $\neg(a(d) \in [\![\alpha_1]\!]^\eta)$ iff (inductive hypothesis) $\neg[\![\mathsf{Cond}_l(a(v), \alpha_1)\ t_\eta[v := t_d]]\!]$ iff (monitor construction) $[\![\mathsf{Cond}_l(a(v), \neg\alpha_1)\ t_\eta[v := t_d]]\!]$ iff $[\![\mathsf{Cond}_l(a(v), \alpha)\ t_\eta[v := t_d]]\!]$.

- Case $\alpha \equiv \alpha_1 \wedge \alpha_2$, and our statement holds by induction for α_1 and α_2. By definition, we have $a(d) \in [\![\alpha]\!]^\eta$ iff $a(d) \in [\![\alpha_1 \wedge \alpha_2]\!]^\eta$ iff (Def. A.1.9: **Semantics of action formulas**) $a(d) \in [\![\alpha_1]\!]^\eta \cap [\![\alpha_2]\!]^\eta$ iff (set theory) $a(d) \in [\![\alpha_1]\!]^\eta \wedge a(d) \in [\![\alpha_2]\!]^\eta$ iff (inductive hypothesis) $[\![\mathsf{Cond}_l(a(v), \alpha_1)\ t_\eta[v := t_d]]\!] \wedge [\![\mathsf{Cond}_l(a(v), \alpha_2)\ t_\eta[v := t_d]]\!]$ holds iff (monitor construction)
  $[\![\mathsf{Cond}_l(a(v), \alpha_1 \wedge \alpha_2)\ t_\eta[v := t_d]]\!]$ holds iff $[\![\mathsf{Cond}_l(a(v), \alpha)\ t_\eta[v := t_d]]\!]$.
- Case $\alpha \equiv \exists d' : D.\alpha_1$, and our statement holds by induction for all η and α_1. By definition, we have $a(d) \in [\![\alpha]\!]^\eta$ iff $a(d) \in [\![\exists d' : D.\alpha_1]\!]^\eta$ iff (Def. A.1.9: **Semantics of action formulas**) $a(d) \in \bigcup_{d' \in D} [\![\alpha_1]\!]^{\eta[d'/v']}$ iff (set theory) $a(d) \in [\![\alpha_1]\!]^{\eta[d'/v']}$ for some $d' \in D$ iff (inductive hypothesis)
  $[\![\mathsf{Cond}_l(a(v), \alpha_1)\ t_\eta[v := t_d]]\!]^{\eta[d'/v']}$ holds for some $d' \in D$ iff
  $\exists d' : D.[\![\mathsf{Cond}_l(a(d), \alpha_1)\ t_\eta[v := t_d]]\!]$ iff (monitor construction)
  $[\![\mathsf{Cond}_l(a(v), \exists d' : D.\alpha_1)\ t_\eta[v := t_d]]\!]$ iff $[\![\mathsf{Cond}_l(a(v), \alpha)\ t_\eta[v := t_d]]\!]$.

□

Lemma 2. Let η be an arbitrary valuation function for data variables, and let t_η be a substitution defined as $t_\eta(v) = t_d$ iff $\eta(v) = d$ for $d \in M_D$. For all regular formulas R, we have that the path $\rho \in [\![R]\!]^\eta$ iff $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho} \surd$.

Proof. We use a structural induction on the structure of the regular formula R. Let η be arbitrary.

- Case $R \equiv \alpha$. By Lemma 1, we have $a(d) \in [\![\alpha]\!]^\eta$ iff $[\![\mathsf{Cond}_l(a(v), \alpha)\ t_\eta[v := t_d]]\!]$ holds, iff (Def. A.1.8: **Semantics of a process**)
  $[\![\bigoplus_{a \in Act} \sum v : D.\ \mathsf{Cond}_l(a(v), \alpha) \to a(v)]\!]^\eta \xrightarrow{a(d)} \surd$ iff (monitor construction)
  $\mathsf{TrR}_l(\alpha)\ t_\eta \xrightarrow{a(d)} \surd$
  Induction hypothesis: For any valuation η, it holds that $\rho \in [\![R]\!]^\eta$ iff
  $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho} \surd$.
- Case $R \equiv R_1 \cdot R_2$, and we have, by our inductive hypothesis, that the statement holds for R_1 and R_2. Observe that we have $\rho \in [\![R]\!]^\eta$ iff $\rho \in [\![R_1 \cdot R_2]\!]^\eta$ iff (Def. A.1.10: **Semantics of regular formulas**) $\exists \rho_1, \rho_2$ such that $\rho = \rho_1 \cdot \rho_2$ and $\rho_1 \in [\![R_1]\!]^\eta, \rho_2 \in [\![R_2]\!]^\eta$ iff (inductive hypothesis) $\exists \rho_1, \rho_2$ such that $\rho = \rho_1 \cdot \rho_2$ and $\mathsf{TrR}_l(R_1)\ t_\eta \xrightarrow{\rho_1} \surd$ and $\mathsf{TrR}_l(R_2)\ t_\eta \xrightarrow{\rho_2} \surd$ iff (Def. A.1.8:**Semantics of a process**) $\mathsf{TrR}_l(R_1)\ t_\eta \xrightarrow{\rho_1} \mathsf{TrR}_l(R_2)\ t_\eta \xrightarrow{\rho_2} \surd$ iff
  $\mathsf{TrR}_l(R_1)\ t_\eta \cdot \mathsf{TrR}_l(R_2)\ t_\eta \xrightarrow{\rho_1 \cdot \rho_2} \surd$ iff $(\mathsf{TrR}_l(R_1) \cdot \mathsf{TrR}_l(R_2))\ t_\eta \xrightarrow{\rho_1 \cdot \rho_2} \surd$

iff (monitor construction) $\mathsf{TrR}_l(R_1 \cdot R_2)\ t_\eta \xrightarrow{\rho_1 \cdot \rho_2} \surd$ iff $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho} \surd$.

- Case $R \equiv R_1 + R_2$, and we have, by our inductive hypothesis, that the statement holds for R_1 and R_2. Observe that we have $\rho \in [\![R]\!]^\eta$ iff $\rho \in [\![R_1 + R_2]\!]^\eta$ iff (Def. A.1.10: **Semantics of regular formulas**) $\rho \in [\![R_1]\!]^\eta \cup [\![R_2]\!]^\eta$ iff (set theory) $\rho \in [\![R_1]\!]^\eta$ or $\rho \in [\![R_2]\!]^\eta$ iff (inductive hypothesis) $\mathsf{TrR}_l(R_1)\ t_\eta \xrightarrow{\rho} \surd$ or $\mathsf{TrR}_l(R_2)\ t_\eta \xrightarrow{\rho} \surd$ iff (Def. A.1.8: **Semantics of a process**)

  $(\mathsf{TrR}_l(R_1) + \mathsf{TrR}_l(R_2))\ t_\eta \xrightarrow{\rho} \surd$ iff (monitor construction)

  $\mathsf{TrR}_l(R_1 + R_2)\ t_\eta \xrightarrow{\rho} \surd$ iff $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho} \surd$.

□

Lemma 3. Let η an arbitrary valuation function, and let t_η be a substitution defined as $t_\eta(v) = t_d$ iff $\eta(v) = d$ for $d \in M_D$. For all regular formulas R, we have that the path $\rho \in [\![R^+]\!]^\eta$ iff $X(l)\ t_\eta \xrightarrow{\rho} \surd$, where $X(l) = \mathsf{TrR}_l(R) \cdot X(l) + \mathsf{TrR}_l(R)$.

Proof. We use structural induction on the number of segments in ρ. Let η, R be arbitrary. We use the equality $R^+ = R \cdot R^+ + R$.

$\rho \in [\![R^+]\!]^\eta$ iff (Def. A.1.10: **Semantics of regular formulas**) $\rho \in [\![R \cdot R^+]\!]^\eta \cup [\![R]\!]^\eta$. Then the base case follows directly from Lemma 2.

Induction hypothesis: Assume that the lemma holds for ρ, i.e., $\rho \in [\![R^+]\!]^\eta$ iff $X(l)\ t_\eta \xrightarrow{\rho} \surd$.

Let $\rho' = \rho'' \cdot \rho$. We reason as follows: $\rho' \in [\![R^+]\!]^\eta$ iff $\rho' \in [\![R \cdot R^+]\!]^\eta \cup [\![R]\!]^\eta$ iff (Lemma 2) $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho'} \surd$ or $\rho' \in [\![R \cdot R^+]\!]^\eta$ iff (Def. A.1.10: **Semantics of regular formulas**) $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho'} \surd$ or $\exists \rho'', \rho$ such that $\rho' = \rho'' \cdot \rho$ and $\rho'' \in [\![R]\!]^\eta, \rho \in [\![R^+]\!]^\eta$ iff (inductive hypothesis, Lemma 2) $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho'} \surd$ or $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho''} \surd$ and $X(l) \xrightarrow{\rho} \surd$ iff (Def. A.1.8: **Semantics of a process**) $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho'} \surd$ or $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho''} X(l)\ t_\eta \xrightarrow{\rho} \surd$ iff $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho'} \surd$ or $\mathsf{TrR}_l(R)\ t_\eta \cdot X(l)\ t_\eta \xrightarrow{\rho'' \cdot \rho} \surd$ iff $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho'} \surd$ or $(\mathsf{TrR}_l(R) \cdot X(l))\ t_\eta \xrightarrow{\rho'' \cdot \rho} \surd$ iff $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho'} \surd$ or $(\mathsf{TrR}_l(R) \cdot X(l))\ t_\eta \xrightarrow{\rho'} \surd$ iff $(\mathsf{TrR}_l(R) + \mathsf{TrR}_l(R) \cdot X(l))\ t_\eta \xrightarrow{\rho'} \surd$ iff

$X(l)\ t_\eta \xrightarrow{\rho'} \surd$ □

Lemma 4. For all modal μ-calculus formulas ϕ, all valuation functions η, substitutions t_η, and all processes P, we have $[\![P]\!] \notin [\![\phi]\!]^\eta$ iff there exists σ such that $(P||_s \mathsf{TrS}_l(\phi))\ t_\eta \xrightarrow{\sigma} \mathrm{error}$, where $||_s$ is the lock-step synchronization between two processes, synchronizing on all actions except for *error*, i.e.,:

$$P||_s \mathsf{TrS}_l(\phi) = \nabla_{\{error,\ action_1\_found,\ldots,\ action_n\_found\}} (\Gamma_{\{action_1|action_1\_monitor \rightarrow action_1\_found,\ \ldots,\ action_n|action_n\_monitor \rightarrow action_n\_found\}} (P||\mathsf{TrS}_l(\phi)))$$

Proof. We again use an induction on the structure of ϕ.

- Case $\phi \equiv b$. Then $[\![P]\!] \notin [\![\phi]\!]\eta$ iff $[\![P]\!] \notin [\![b]\!]\eta$.

  Let $[\![P]\!] \notin [\![b]\!]\eta$. This implies that $\eta(b)$ must not hold (because otherwise, by Def. A.1.11: **Semantics of modal μ-calculus formulas** we have $[\![b]\!]^\eta = S$). But then $[\![\neg b]\!]^\eta$ holds. We then have from Lemma 1 and 2 that $[\![\neg b \rightarrow \text{error}]\!]^\eta = \text{error}$, from which the required statement follows immediately.

- Case $\phi \equiv \phi_1 \wedge \phi_2$, and assume that the property holds for all processes P and valuation functions η, substitutions t_η, and μ-calculus formulas ϕ_1, ϕ_2. Then we must prove that $[\![P]\!] \notin [\![\phi]\!]\eta$ iff $(P||_s \mathsf{TrS}_l(\phi))\ t_\eta \xrightarrow{\sigma} \text{error}$ for some σ. We reason as follows.

  $[\![P]\!] \notin [\![\phi]\!]^\eta$ iff $[\![P]\!] \notin [\![\phi_1 \wedge \phi_2]\!]^\eta$ iff (Def. A.1.11: **Semantics of modal μ-calculus formulas**) $[\![P]\!] \notin [\![\phi_1]\!]^\eta \cap [\![\phi_2]\!]^\eta$ iff (set theory) $\neg([\![P]\!] \in [\![\phi_1]\!]^\eta \cap [\![\phi_2]\!]^\eta)$ iff $\neg([\![P]\!] \in [\![\phi_1]\!]^\eta \wedge [\![P]\!] \in [\![\phi_2]\!]^\eta)$ iff $\neg([\![P]\!] \in [\![\phi_1]\!]^\eta) \vee \neg([\![P]\!] \in [\![\phi_2]\!]^\eta)$ iff $[\![P]\!] \notin [\![\phi_1]\!]\eta$ or $[\![P]\!] \notin [\![\phi_2]\!]\eta$ iff (inductive hypothesis) there exist σ, σ' for which $(P||_s \mathsf{TrS}_l(\phi_1))\ t_\eta \xrightarrow{\sigma} \text{error}$ or $(P||_s \mathsf{TrS}_l(\phi_2))\ t_\eta \xrightarrow{\sigma'} \text{error}$ iff (Def.A.1.8: **Semantics of a process**)

  $(P||_s(\mathsf{TrS}_l(\phi_1) + \mathsf{TrS}_l(\phi_2)))\ t_\eta \xrightarrow{\sigma''} \text{error}$ for some σ'', iff (monitor construction) $(P||_s \mathsf{TrS}_l(\phi_1 \wedge \phi_2))\ t_\eta \xrightarrow{\sigma''} \text{error}$.

- Case $\phi \equiv \forall d : D.\phi_1$, and assume that the property holds for all processes P, and η, t_η, and ϕ_1. Then we must prove that $[\![P]\!] \notin [\![\phi]\!]^\eta$ iff $(P||_s \mathsf{TrS}_l(\phi))\ t_\eta \xrightarrow{\sigma} \text{error}$ for some σ. We reason as follows.

  $[\![P]\!] \notin [\![\phi]\!]\eta$ iff $[\![P]\!] \notin [\![\forall d : D.\phi_1]\!]\eta$ iff (Def. A.1.11: **Semantics of modal μ-calculus formulas**) $[\![P]\!] \notin \bigcap_{d \in D} [\![\phi_1]\!]^{\eta[d/v]}$ iff (set theory) for some $d \in D$, $[\![P]\!] \notin [\![\phi_1]\!]^{\eta[d/v]}$, iff (inductive hypothesis) for some $d \in D$, valuation function η, substitution t_η, and for some σ, $(P||_s \mathsf{TrS}_l(\phi_1))\ t_\eta \xrightarrow{\sigma} \text{error}$ iff (Def. A.1.8: **Semantics of a process**) $(P||_s \sum d : D.\mathsf{TrS}_l(\phi_1))\ t_\eta \xrightarrow{\sigma} \text{error}$ iff (monitor construction) $(P||_s \mathsf{TrS}_l(\forall d : D.\phi_1))\ t_\eta \xrightarrow{\sigma} \text{error}$.

- Case $\phi \equiv [R]\phi_1$ and assume that the property holds for all processes P, and η, t_η, and ϕ_1. Then we must prove that $[\![P]\!] \notin [\![\phi]\!]^\eta$ iff $(P||_s \mathsf{TrS}_l(\phi))\ t_\eta \xrightarrow{\sigma}$ error for some σ. We reason as follows.

  $[\![P]\!] \notin [\![\phi]\!]\eta$ iff $[\![P]\!] \notin [\![[R]\phi_1]\!]\eta$ iff (Def. A.1.11: **Semantics of modal μ-calculus formulas**) $[\![P]\!] \xrightarrow{\rho} [\![P']\!]$ for some $\rho \in [\![R]\!]\eta$, for which $[\![P']\!] \notin [\![\phi_1]\!]\eta$ iff (Lemma 2), $[\![P]\!] \xrightarrow{\rho} [\![P']\!]$ for $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho} \surd$ for which $[\![P']\!] \notin [\![\phi_1]\!]\eta$ iff (inductive hypothesis) $[\![P]\!] \xrightarrow{\rho} [\![P']\!]$ for $\mathsf{TrR}_l(R)\ t_\eta \xrightarrow{\rho} \surd$ for which $(P'\ ||_s \mathsf{TrS}_l(\phi_1))\ t_\eta \xrightarrow{\sigma}$ error for some σ iff (Def. A.1.8: **Semantics of a process**) $(P||_s \mathsf{TrR}_l(R) \cdot \mathsf{TrS}_l(\phi_1))\ t_\eta \xrightarrow{\rho\cdot\sigma}$ error for some σ, ρ iff (monitor construction) $(P||_s \mathsf{TrS}_l([R]\phi_1))\ t_\eta \xrightarrow{\sigma'}$ error for some σ'.

- Case $\phi \equiv [R^*]\phi_1$ and assume that the property holds for all processes P, and η, t_η, and ϕ_1. Then we must prove that $[\![P]\!] \notin [\![\phi]\!]^\eta$ iff $(P||_s \mathsf{TrS}_l(\phi))\ t_\eta \xrightarrow{\sigma}$ error for some σ. We use the equality $R^* = R^+ + \epsilon$, where ϵ represents an empty path (no transition). We reason as follows.

  $[\![P]\!] \notin [\![\phi]\!]^\eta$ iff $[\![P]\!] \notin [\![[R^*]\phi_1]\!]^\eta$ iff (Def. A.1.11: **Semantics of modal μ-calculus formulas**) $[\![P]\!] \xrightarrow{\rho} [\![P']\!]$ for some $\rho \in [\![R^*]\!]^\eta$, for which $[\![P']\!] \notin [\![\phi_1]\!]^\eta$ iff $[\![P]\!] \xrightarrow{\rho} [\![P']\!]$ for $\rho \in [\![R^+]\!]^\eta$ or $\rho = \epsilon$, and for which $[\![P']\!] \notin [\![\phi_1]\!]^\eta$ iff (Lemma 3) $[\![P]\!] \xrightarrow{\rho} [\![P']\!]$ for $X(l)\ t_\eta \xrightarrow{\rho} \surd$, or for $\rho = \epsilon$ for which $[\![P']\!] \notin [\![\phi_1]\!]^\eta$ iff (inductive hypothesis) $[\![P]\!] \xrightarrow{\rho} [\![P']\!]$ for $X(l)\ t_\eta \xrightarrow{\rho} \surd$ or $\rho = \epsilon$ for which $(P'||_s \mathsf{TrS}_l(\phi_1))\ t_\eta \xrightarrow{\sigma}$ error for some σ iff (Def. A.1.8: **Semantics of a process**) $(P||_s(\mathsf{TrS}_l(\phi_1) + X(l) \cdot \mathsf{TrS}_l(\phi_1)))\ t_\eta \xrightarrow{\rho\cdot\sigma}$ error for some σ, ρ iff (monitor construction) $(P||_s \mathsf{TrS}_l([R^*]\phi_1))\ t_\eta \xrightarrow{\sigma'}$ error for some σ'.

□

$$\frac{}{\alpha \xrightarrow{[\![\alpha]\!]} \surd}$$

$$\frac{p \xrightarrow{\omega} \surd}{p + q \xrightarrow{\omega} \surd} \qquad \frac{p \xrightarrow{\omega} p'}{p + q \xrightarrow{\omega} p'}$$

$$\frac{q \xrightarrow{\omega} \surd}{p + q \xrightarrow{\omega} \surd} \qquad \frac{q \xrightarrow{\omega} q'}{p + q \xrightarrow{\omega} q'}$$

$$\frac{p \xrightarrow{\omega} \surd}{p \cdot q \xrightarrow{\omega} q} \qquad \frac{p \xrightarrow{\omega} p'}{p \cdot q \xrightarrow{\omega} p' \cdot q}$$

$$\frac{p \xrightarrow{\omega} \surd}{c \to p \xrightarrow{\omega} \surd}\ [\![c]\!] = \text{true} \qquad \frac{p \xrightarrow{\omega} p'}{c \to p \xrightarrow{\omega} p'}\ [\![c]\!] = \text{true}$$

$$\frac{p \xrightarrow{\omega} \surd}{c \to p \diamond q \xrightarrow{\omega} \surd}\ [\![c]\!] = \text{true} \qquad \frac{p \xrightarrow{\omega} p'}{c \to p \diamond q \xrightarrow{\omega} p'}\ [\![c]\!] = \text{true}$$

$$\frac{q \xrightarrow{\omega} \surd}{c \to p \diamond q \xrightarrow{\omega} \surd}\ [\![c]\!] = \text{false} \qquad \frac{q \xrightarrow{\omega} q'}{c \to p \diamond q \xrightarrow{\omega} q'}\ [\![c]\!] = \text{false}$$

Table A.1: Structural Operational Semantics for the basic operators

$$\frac{p[t_e/d] \xrightarrow{\omega} \surd}{\sum_{d:D} p \xrightarrow{\omega} \surd}\ e \in M_D \qquad \frac{p[t_e/d] \xrightarrow{\omega} p'}{\sum_{d:D} p \xrightarrow{\omega} p'}\ e \in M_D$$

Table A.2: Structural Operational Semantics for the sum operator

$$\frac{p \xrightarrow{\omega} \surd}{p \| q \xrightarrow{\omega} q} \qquad \frac{p \xrightarrow{\omega} p'}{p \| q \xrightarrow{\omega} p' \| q}$$

$$\frac{q \xrightarrow{\omega} \surd}{p \| q \xrightarrow{\omega} p} \qquad \frac{q \xrightarrow{\omega} q'}{p \| q \xrightarrow{\omega} p \| q'}$$

$$\frac{p \xrightarrow{\omega} \surd, q \xrightarrow{\overline{\omega}} q'}{p \| q \xrightarrow{\omega|\overline{\omega}} q'} \qquad \frac{p \xrightarrow{\omega} p', q \xrightarrow{\overline{\omega}} \surd}{p \| q \xrightarrow{\omega|\overline{\omega}} p'}$$

$$\frac{p \xrightarrow{\omega} \surd, q \xrightarrow{\overline{\omega}} \surd}{p \| q \xrightarrow{\omega|\overline{\omega}} \surd} \qquad \frac{p \xrightarrow{\omega} p', q \xrightarrow{\overline{\omega}} q'}{p \| q \xrightarrow{\omega|\overline{\omega}} p' \| q'}$$

Table A.3: Structural Operational Semantics for the parallel operator

$$\frac{p \xrightarrow{\omega} \surd}{p \between q \xrightarrow{\omega} q} \qquad \frac{p \xrightarrow{\omega} p'}{p \between q \xrightarrow{\omega} p' \| q}$$

$$\frac{p \xrightarrow{\omega} \surd, q \xrightarrow{\overline{\omega}} q'}{p | q \xrightarrow{\omega|\overline{\omega}} q'} \qquad \frac{p \xrightarrow{\omega} p', q \xrightarrow{\overline{\omega}} \surd}{p | q \xrightarrow{\omega|\overline{\omega}} p'}$$

$$\frac{p \xrightarrow{\omega} \surd, q \xrightarrow{\overline{\omega}} \surd}{p | q \xrightarrow{\omega|\overline{\omega}} \surd} \qquad \frac{p \xrightarrow{\omega} p', q \xrightarrow{\overline{\omega}} q'}{p | q \xrightarrow{\omega|\overline{\omega}} p' \| q'}$$

Table A.4: Structural Operational Semantics for the auxiliary parallel operators

$$\frac{p \xrightarrow{\omega} \surd}{\nabla_V(p) \xrightarrow{\omega} \surd}\ \underline{\omega} \in V \cup \{\tau\} \qquad \frac{p \xrightarrow{\omega} p'}{\nabla_V(p) \xrightarrow{\omega} \nabla_V(p')}\ \underline{\omega} \in V \cup \{\tau\}$$

$$\frac{p \xrightarrow{\omega} \surd}{\partial_B(p) \xrightarrow{\omega} \surd}\ \underline{\omega}\{\} \cap B = \emptyset \qquad \frac{p \xrightarrow{\omega} p'}{\partial_B(p) \xrightarrow{\omega} \partial_B(p')}\ \underline{\omega}\{\} \cap B = \emptyset$$

$$\frac{p \xrightarrow{\omega} \surd}{\rho_R(p) \xrightarrow{R \bullet \omega} \surd} \qquad \frac{p \xrightarrow{\omega} p'}{\rho_R(p) \xrightarrow{R \bullet \omega} \rho_R(p')}$$

$$\frac{p \xrightarrow{\omega} \surd}{\Gamma_C(p) \xrightarrow{\gamma_C(\omega)} \surd} \qquad \frac{p \xrightarrow{\omega} p'}{\Gamma_C(p) \xrightarrow{\gamma_C(\omega)} \Gamma_C(p')}$$

$$\frac{p \xrightarrow{\omega} \surd}{\tau_I(p) \xrightarrow{\theta_I(\omega)} \surd} \qquad \frac{p \xrightarrow{\omega} p'}{\tau_I(p) \xrightarrow{\theta_I(\omega)} \tau_I(p')}$$

Table A.5: Structural Operational Semantics for the auxiliary operators

$$\frac{q[t_1/d_1, \ldots, t_n/d_n] \xrightarrow{\omega} \surd}{X(t_1, \ldots, t_n) \xrightarrow{\omega} \surd} \qquad \frac{q[t_1/d_1, \ldots, t_n/d_n] \xrightarrow{\omega} q'}{X(t_1, \ldots, t_n) \xrightarrow{\omega} q'}$$

where $X(d_1 : D_1, \ldots, d_n : D_n) = q \in PE$

$$\frac{q[u_1/d_1, \ldots, u_n/d_n] \xrightarrow{\omega} \surd}{X(d'_1 = t_1, \ldots, d'_m = t_m) \xrightarrow{\omega} \surd} \qquad \frac{q[u_1/d_1, \ldots, u_n/d_n] \xrightarrow{\omega} q'}{X(d'_1 = t_1, \ldots, d'_m = t_m) \xrightarrow{\omega} q'}$$

where $X(d_1 : D_1, \ldots, d_n : D_n) = q \in PE$ and

$$u_i = \begin{cases} \xi(t_j) & \text{if } d_i = d'_j \text{ for some } 1 \leq j \leq m, \\ \xi(d_i) & \text{otherwise.} \end{cases}$$

Table A.6: Structural Operational Semantics for recursion

References

[1] Large Hadron Collider beauty experiment. URL http://lhcb-public.web.cern.ch/lhcb-public.

[2] CodeSurfer. URL http://www.grammatech.com/products/codesurfer.

[3] Frama-C software suite. URL http://frama-c.com.

[4] mCRL2 SVN Repository. URL https://svn.win.tue.nl/viewvc/MCRL2.

[5] Worldwide LHC Computing Grid, 2015. URL http://www.cern.ch/about/computing.

[6] Silvia Abrahao et al. Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Transactions on Software Engineering*, 39(3):327–342, 2013.

[7] Jörg Ackermann and Klaus Turowski. A library of OCL specification patterns for behavioral specification of software components. In *Advanced Information Systems Engineering*, pages 255–269. Springer, 2006.

[8] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. *Advances in Database Technology—EDBT'98*, pages 467–483, 1998.

[9] Cristina Aiftimiei et al. Design and implementation of the gLite CREAM job management service. *Future Generation Computer Systems*, 26(4):654–667, 2010.

[10] Behzad Akbarpour, Amr T. Abdel-Hamid, Sofiène Tahar, and John Harrison. Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. *The Computer Journal*, 53(4):465–488, 2010.

[11] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[12] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02. ACM.

[13] Roy Armoni et al. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002,*, pages 296–211, 2002. doi: 10.1007/3-540-46002-0_21. URL http://dx.doi.org/10.1007/3-540-46002-0_21;http://dblp.uni-trier.de/rec/bib/conf/tacas/ArmoniFFGGKLMSTVZ02.

[14] Silky Arora, Ambar Gadkari, and Sethu Ramesh. Scenario-Based Specification of Automotive Requirements With Quantitative Constraints and Synthesis of SL/SF Monitors. *Embedded Systems Letters, IEEE*, 3(2):62–65, 2011.

[15] Marco Autili, Paola Inverardi, and Patrizio Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering*, 14(3):293–340, 2007.

[16] Bahareh Badban et al. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[17] Jos C.M. Baeten, Twan Basten, and Michel A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50. Cambridge University Press, 2009.

[18] Thomas Ball et al. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods*, pages 1–20. Springer, 2004.

[19] Aritra Bandyopadhyay and Sudipto Ghosh. Test input generation using UML sequence and state machines models. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 121–130. IEEE, 2009.

[20] Luciano Baresi, Carlo Ghezzi, and Luca Zanolin. Modeling and validation of publish/subscribe architectures. In *Testing Commercial-off-the-shelf Components and Systems*, pages 273–291. Springer, 2005.

[21] Andreas Bauer. Monitorability of ω-regular languages. *arXiv preprint arXiv:1006.3638*, 2010.

[22] Dmitry Bedrin. jSonde: Profiler and reverse engineering tool for Java, 2015. URL https://bedrin.github.io/jsonde.

[23] Maurice H. Beek and Erik P. Vink. Towards Modular Verification of Software Product Lines with mCRL2. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Tech-*

nologies for Mastering Change, volume 8802, pages 368–385, 2014.

[24] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109–137, 1984.

[25] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 35–45. ACM, 2002.

[26] Dirk Beyer et al. The Software Model Checker BLAST: Applications to Software Engineering. *International Journal on Software Tools Technology Transfer*, 9 (5):505–525, 2007.

[27] Ian Bird. Computing for the large hadron collider. *Annual Review of Nuclear and Particle Science*, 61:99–118, 2011.

[28] Dines Bjørner and Klaus Havelund. 40 years of formal methods. In *FM 2014: Formal Methods*, pages 42–61. Springer, 2014.

[29] Stefan Blom and Jaco van de Pol. Symbolic Reachability for Process Algebras with Recursive Data Types. In *5th Int. Colloq. Theoretical Aspects of Computing*, volume 5160 of *LNCS*, pages 81–95, 2008.

[30] Dragan Bošnački, Aad Mathijssen, and Yaroslav S. Usenko. Behavioural analysis of an I2C Linux driver. In *Formal Methods for Industrial Critical Systems*, pages 205–206. Springer, 2009.

[31] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods... ten years later. *Computer*, 39(1):40–48, 2006.

[32] Julian Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL using observational mu-calculus. In *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 203–217. Springer, 2002.

[33] Niels C. W. M. Braspenning, E. M. Bortnik, Joanna M. van de Mortel-Fronczak, and Jacobus E. Rooda. Model-based System Analysis Using Chi and Uppaal: An Industrial Case Study. *Computers in Industry*, 59(1):41–54, 2008.

[34] Guillaume Brat and Willem Visser. Combining static analysis and model checking for software analysis. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering*, pages 262–269. IEEE, 2001.

[35] Lionel C Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[36] Guy H. Broadfoot. ASD case notes: Costs and benefits of applying formal methods to industrial control software. In *FM 2005: Formal Methods*, pages 548–551. Springer, 2005.

[37] Manfred Broy et al. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In *Models in Software Engineering*, pages 318–323. Springer, 2007.

[38] Jerry R. Burch et al. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society, 1990.

[39] Honghua Cao, Shi Ying, and Dehui Du. Towards model-based verification of BPEL with model checking. In *Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*, pages 190–190. IEEE, 2006.

[40] Jan Cederquist, Ricardo Corin, and Mohammad T. Dashti. On the quest for impartiality: design and analysis of a fair non-repudiation protocol. In *Proceedings of the 7th international conference on Information and Communications Security*, pages 27–39. Springer-Verlag, 2005.

[41] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: an industrial case study. In *Proceedings of the 24th International Conference on Software Engineering*, pages 431–441. ACM, 2002.

[42] Bo Chen and Samira Sadaoui. Simulation and Verification of a Dynamic Online Auction. In *Proc. 7th Int. Conf. Software Eng. and Applications*, pages 385–390, 2003.

[43] Cosmina Chişe and Ioan Jurca. Towards early performance assessment based on UML MARTE models for distributed systems. In *Proceedings of the 5th International Symposium on Applied Computational Intelligence and Informatics*, pages 521–526. IEEE, 2009.

[44] Alessandro Cimatti et al. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 359–364. Springer, 2002.

[45] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[46] Edmund M Clarke and Anca I. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 428–437. Springer, 1989.

[47] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchro-

nization skeletons using branching time temporal logic. *Logics of Programs*, pages 52–71, 1982.

[48] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[49] Edmund M. Clarke, E. Allen Emerson, and Aravinda P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8 (2):244–263, 1986.

[50] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11): 74–84, 2009.

[51] Rachel L. Cobleigh, George S. Avrunin, and Lori A. Clarke. User guidance for creating precise and accessible property specifications. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 208–218. ACM, 2006.

[52] Séverine Colin and Leonardo Mariani. Run-Time Verification. In *Model-Based Testing of Reactive Systems*, pages 525–555. Springer, 2005.

[53] James C. Corbett et al. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 439–448. IEEE, 2000.

[54] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2):103–179, 1992.

[55] Sjoerd Cranen, Jan Friso Groote, and Michel Reniers. A linear translation from CTL* to the first-order modal μ-calculus. *Theoretical Computer Science*, 412(28):3129–3139, 2011.

[56] Sjoerd Cranen, Bas Luttik, and Tim A.C. Willemse. Evidence for fixpoint Logic. *Proceedings of the 24th EACSL Annual Conference on Computer Science Logic*, to appear, 2015.

[57] Sjoerd Cranen et al. An overview of the mCRL2 toolset and its recent advances. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.

[58] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972.

[59] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[60] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, pages 407–419. Springer, 1990.

[61] Maria del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Debugging UML designs with model checking. *Journal of Object Technology*, 1(2):101–117, 2002.

[62] Volker Diekert, Anca Muscholl, and Igor Walukiewicz. A Note on Monitors and Büchi automata. *CoRR*, abs/1507.01020, 2015.

[63] Laura K. Dillon. Translation of Future Interval Logic into Linear-Time Temporal Logic. Technical report, Department of Computer Science, Michigan State University, September 1999.

[64] Zinovy Diskin and Juergen Dingel. Mappings, maps and tables: Towards formal semantics for associations in UML2. In *Model Driven Engineering Languages and Systems*, pages 230–244. Springer, 2006.

[65] Salvatore Distefano, Marco Scarpa, and Antonio Puliafito. From UML to Petri nets: The PCM-based methodology. *IEEE Transactions on Software Engineering*, 37(1):65–79, 2011.

[66] Jori Dubrovin and Tommi Junttila. Symbolic model checking of hierarchical UML state machines. In *Proceedings of the 8th International Conference on Application of Concurrency to System Design*, pages 108–117. IEEE, 2008.

[67] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM, 1998.

[68] Technische Universiteit Eindhoven. mCRL2 Showcases. URL http://www.mcrl2.org/release/user_manual/showcases.html.

[69] William Elmendorf. Evaluation of the Functional Testing of Control Programs. Technical report, IBM Corporation Systems Development Division, 1967.

[70] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[71] Gregor Engels et al. From UML activities to TAAL-towards behaviour-preserving model transformations. In *Model Driven Architecture–Foundations and Applications*, pages 94–109. Springer, 2008.

[72] Alessandro Fantechi et al. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4(3):243–263,

1994.

[73] Borja Fernandez Adiego et al. Bringing Automated Model Checking to PLC Program Development-A CERN Case Study. Technical report, 2014.

[74] Predrag Filipovikj, Mattias Nyberg, and Guillermo Rodriguez-Navas. Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In *Proceedings of the 22nd International Requirements Engineering Conference*, pages 444–450. IEEE, 2014.

[75] Stephan Flake and Wolfgang Mueller. Formal semantics of static and temporal state-oriented OCL constraints. *Software and Systems Modeling*, 2(3):164–186, 2003.

[76] Markus Fockel and Jorg Holtmann. A requirements engineering methodology combining models and controlled natural language. In *Proceedings of the 4th International Model-Driven Requirements Engineering Workshop*, pages 67–76. IEEE, 2014.

[77] Wan Fokkink. *Modelling distributed systems*. Springer Science & Business Media, 2007.

[78] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[79] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000.

[80] Hubert Garavel and Holger Hermanns. On combining functional verification and performance evaluation using CADP. In *FME 2002: Formal Methods—Getting IT Right*, pages 410–429. Springer, 2002.

[81] Hubert Garavel et al. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[82] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering*, pages 412–416. IEEE, 2001.

[83] Jan Friso Groote and Radu Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, pages 74–90. Springer-Verlag, 1999.

[84] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and analysis of communicating systems*. MIT Press, 2014.

[85] Jan Friso Groote and Tim A.C. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56(3):251–273, 2005.

[86] Jan Friso Groote, Ammar Osaiweran, and Jacco H Wesselius. Analyzing the effects of formal methods on the development of industrial control software. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pages 467–472. IEEE, 2011.

[87] Jan Friso. Groote et al. The Formal Specification Language mCRL2. In *Methods for Modelling Software Systems*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), 2007.

[88] Jan Friso Groote et al. Experiences in developing the mCRL2 toolset. *Software: Practice and Experience*, 41(2):143–153, 2011.

[89] Radu Grosu et al. Safety-liveness semantics for UML 2.0 sequence diagrams. In *Proceedings of the 5th International Conference on Application of Concurrency to System Design*, pages 6–14. IEEE, 2005.

[90] Object Management Group. UML Specifications. URL http://www.omg.org/spec.

[91] Object Management Group. UML 2.4 Infrastructure Specification, 2012. URL http://www.omg.org/spec/UML/2.4/Infrastructure.

[92] Object Management Group. UML 2.4 Superstructure Specification, 2012. URL http://www.omg.org/spec/UML/2.4/Superstructure.

[93] Object Management Group. The UML profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems, 2013. URL http://www.omgmarte.org/.

[94] Yann-Gaël Guéhéneuc and Tewfik Ziadi. Automated reverse-engineering of UML v2. 0 dynamic models. In *Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering*. Springer, 2005.

[95] Nicolas Guelfi and Amel Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 283–290. IEEE Computer Society, 2005.

[96] Qiang Guo. Verifying Erlang/OTP components in μCRL. In *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems*, pages 227–246. Springer, 2007.

[97] Qiang Guo, John Derrick, and Csaba Hoch. Verifying Erlang telecommunication systems with the process algebra μCRL. In *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems*, pages 201–217. Springer, 2008.

[98] Arie Gurfinkel and Marsha Chechik. Multi-valued model checking via classical model checking. In *CONCUR 2003-Concurrency Theory*, pages 266–280. Springer, 2003.

[99] Helle Hvid Hansen et al. Automated verification of executable UML models. In *Proceedings of the 9th international conference on Formal Methods for Components and Objects*, pages 225–250. Springer-Verlag, 2010.

[100] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software & Systems Modeling*, 7(2):237–252, 2008.

[101] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[102] Øystein Haugen et al. STAIRS towards formal design with sequence diagrams. *Software & Systems Modeling*, 4(4):355–357, 2005.

[103] Klaus Havelund. Runtime Verification of C Programs. *Testing of Software and Communicating Systems*, pages 7–22, 2008.

[104] Yu-Tong He and Ryszard Janicki. Verifying protocols by model checking: a case study of the wireless application protocol and the model checker SPIN. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative Research*, pages 174–188. IBM Press, 2004.

[105] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[106] Hossein Hojjat, Mohammad Reza Mousavi, and Marjan Sirjani. Formal analysis of SystemC designs in process algebra. *Fundamenta Informaticae*, 107(1): 19–42, 2011.

[107] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall Inc., 1990.

[108] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[109] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *SPIN Model Checking and Software Verification*, pages 131–147. Springer, 2000.

[110] Juraj Hromkovic, Sebastian Seibert, and Thomas Wilke. Translating Regular Expressions into Small epsilon-Free Nondeterministic Finite Automata. In *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, pages 55–66. Springer-Verlag, 1997.

[111] Yi-Ling Hwong, Vincent Kusters, and Tim A.C. Willemse. Analysing the Control Software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. *Fundamentals of Software Engineering*, pages 174–189, 2012.

[112] Inria. Case Studies Achieved using the CADP Toolset - Inria. URL http://cadp.inria.fr/case-studies.

[113] Syed M.S. Islam, Mohammed H. Sqalli, and Sohel Khan. Modeling and Formal Verification of DHCP Using SPIN. *International Journal of Computer Science and Applications*, 3(2):145–159, 2006.

[114] Wil Janssen et al. Model checking for managers. In *Theoretical and Practical Aspects of SPIN model checking*, pages 92–107. Springer, 1999.

[115] Toni Jussila et al. Model checking dynamic and hierarchical UML state machines. In *Proceedings of the 3rd International Workshop on Model Development, Validation and Verification*, pages 94–110, 2006.

[116] Gijs Kant and Jaco van de Pol. Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games. In *Proceedings of the First Workshop on GRAPH Inspection and Traversal Engineering*, pages 50–65, 2012.

[117] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink. Defining Object-oriented Execution Semantics Using Graph Transformations. In *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 186–201, 2006.

[118] Gerwin Klein et al. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[119] Alexander Knapp and Jochen Wuttke. Model checking of UML 2.0 interactions. In *Models in Software Engineering*, pages 42–51. Springer, 2007.

[120] Sascha Konrad and Betty H.C. Cheng. Facilitating the construction of specification pattern-based properties. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 329–338. IEEE, 2005.

[121] Hillel Kugler et al. Temporal logic for scenario-based specifications. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–460. Springer, 2005.

[122] Juliana Küster-Filipe. Modelling concurrent interactions. *Theoretical Computer*

Science, 351(2):203–220, 2006.

[123] Alfons Laarman, Jaco van de Pol, and Michael Weber. Multi-core LTSmin: Marrying modularity and scalability. In *NASA Formal Methods*, pages 506–511. Springer, 2011.

[124] Leslie Lamport. Sometime is sometimes not never: On the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185. ACM, 1980.

[125] Leslie Lamport. Proving possibility properties. *Theoretical Computer Science*, 206(1):341–352, 1998.

[126] Erwin Laure et al. Programming the Grid with gLite. *Computational Methods in Science and Technology*, 12(1):33–45, 2006.

[127] Codrut L. Lazăr et al. Tool Support for fUML Models. *International Journal of Computers, Communications & Control*, 5(5):775–782, 2010.

[128] Insup Lee and Oleg Sokolsky. A graphical property specification language. In *Proceedings of the 2nd IEEE Workshop on High-Assurance Systems Engineering*, pages 42–47. IEEE, 1997.

[129] Johan Lilius and Ivan Porres Paltor. vUML: A tool for verifying UML models. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 255–258. IEEE, 1999.

[130] Johan Lilius and Ivan Porres Paltor. Formalising UML state machines for model checking. In *«UML»'99—The Unified Modeling Language*, pages 430–444. Springer, 1999.

[131] Vitor Lima et al. Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. *Electronic Notes in Theoretical Computer Science*, 254:143–160, 2009.

[132] Jian-Guang Lou et al. Mining program workflow from interleaved traces. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 613–622. ACM, 2010.

[133] Madjid Maidi. The common fragment of CTL and LTL. In *Proceedings. 41st Annual Symposium on Foundations of Computer Science*, pages 643–652. IEEE, 2000.

[134] Radu Mateescu. Property Pattern Mappings for Regular Alternation-Free μ-Calculus. URL http://www.inrialpes.fr/vasy/cadp/resources/evaluator/rafmc.html.

[135] Radu Mateescu. Specification and Analysis of Asynchronous Systems using CADP. *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*, pages 141–169, 2010.

[136] Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free μ-calculus. *Science of Computer Programming*, 46(3): 255–281, 2003.

[137] Aad Mathijssen and A. Johannes Pretorius. Specification, Analysis and Verification of an Automated Parking Garage. Technical report, Eindhoven University of Technology, The Netherlands, 2005.

[138] Aad Mathijssen and A. Johannes Pretorius. Verified design of an automated parking garage. In *Formal Methods: Applications and Technology*, pages 165–180. Springer, 2007.

[139] Olga Shumsky Matlin, Ewing L. Lusk, and William McCune. SPINning Parallel Systems Software. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 213–220. Springer-Verlag, 2002.

[140] John William McManus and William L Bynum. Design and analysis techniques for concurrent blackboard systems. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 26(6):669–680, 1996.

[141] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software & Systems Modeling*, 10(4):489–514, 2011.

[142] Oscar Mondragón, Ann Q. Gates, and Steven Roach. Prospec: Support for elicitation and formal specification of software properties. *Electronic Notes in Theoretical Computer Science*, 89(2):67–88, 2003.

[143] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[144] Sam Owre, John M Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, pages 748–752. Springer-Verlag, 1992.

[145] Karl Palmskog. Verification of the session management protocol. *Royal Institute of Technology, Masters Thesis*, 2006.

[146] David Lorge Parnas. Really Rethinking 'Formal Methods'. *IEEE Computer*, 43: 28–34, 2010.

[147] Stuart K. Paterson and Andrei Tsaregorodtsev. Dirac Optimized Workload Management. *Journal of Physics: Conference Series*, 119(6), 2008.

[148] Radek Pelánek. Fighting state space explosion: Review and evaluation. In *Formal Methods for Industrial Critical Systems*, pages 37–52. Springer, 2009.

[149] Willem Penninckx et al. Sound formal verification of Linux's USB BP keyboard driver. In *NASA Formal Methods*, pages 210–215. Springer, 2012.

[150] Dorina C. Petriu. Software Model-based Performance Analysis. *Model-Driven Engineering for Distributed Real-Time Systems: MARTE Modeling, Model Transformations and their Usages*, pages 139–166, 2010.

[151] Dorina C. Petriu and Hui Shen. Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications. In *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 159–177. Springer-Verlag, 2002.

[152] Simon Pickin et al. System test synthesis from UML models of distributed software. In *Formal Techniques for Networked and Distributed Sytems*, pages 97–113. Springer, 2002.

[153] Bas Ploeger. Analysis of ACS using mCRL2. *Technische Universiteit Eindhoven, CS-Report*, pages 09–11, 2009.

[154] Katerina Pokozy-Korenblat and Corrado Priami. Toward extracting π-calculus from UML sequence and state diagrams. *Electronic Notes in Theoretical Computer Science*, 101:51–72, 2004.

[155] Amalinda Post et al. Automotive behavioral requirements expressed in a specification pattern system: a case study at BOSCH. *Requirements Engineering*, 17(1):19–33, 2012.

[156] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[157] Adrian Casajus Ramo and Ricardo Graciani Diaz. Executor Framework for DIRAC. *Journal of Physics: Conference Series*, 396(5):052020, 2012.

[158] Adrian Casajus Ramo et al. DIRAC pilot framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*, 219(6), 2010.

[159] Holger Rasch and Heike Wehrheim. Checking the validity of scenarios in UML models. In *Formal Methods for Open Object-Based Distributed Systems*, pages 67–82. Springer, 2005.

[160] Wolfgang Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer Science & Business Media, 2013.

[161] Daniela Remenska. PASSWebStart: Property ASSistant tool for eliciting μ-calculus properties, 2014. URL https://github.com/remenska/PASSWebStart.

[162] Daniela Remenska and Philip Homburg. The mCRL2⇔UML transformation toolset, 2013. URL https://github.com/remenska/NFM.

[163] Daniela Remenska and Tim A.C. Willemse. PASS: Property ASSistant tool for Eclipse, 2014. URL https://github.com/remenska/PASS.

[164] Daniela Remenska and Tim A.C. Willemse. Extended Property Pattern Mappings for μ-calculus, 2014. URL https://remenska.github.io/patterns.

[165] Daniela Remenska et al. Using model checking to analyze the system behavior of the LHC production grid. *Future Generation Computer Systems*, 29(8):2239–2251, 2013.

[166] Daniela Remenska et al. From UML to process algebra and back: An automated approach to model-checking software design artifacts of concurrent systems. In *NASA Formal Methods*, pages 244–260. Springer, 2013.

[167] Daniela Remenska et al. Property specification made easy: Harnessing the power of model checking in UML designs. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 17–32. Springer, 2014.

[168] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[169] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Control flow analysis for reverse engineering of sequence diagrams. Technical report, Ohio State University, 2014.

[170] Gwen Salaün et al. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Assurances for Self-Adaptive Systems*, pages 60–79. Springer, 2013.

[171] Monalisa Sarma, Debasish Kundu, and Rajib Mall. Automatic test case generation from UML sequence diagrams. In *Proceedings of the 15th International Conference on Advanced Computing and Communications*, volume 60, page 65. IEEE Computer Society, 2007.

[172] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357–369, 2001.

[173] David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *Static Analysis*, pages 351–380. Springer, 1998.

[174] Hui Shen, Aliya Virani, and Jianwei Niu. Formalize UML 2 Sequence Diagrams. In *Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 437–440. IEEE Computer Society, 2008.

[175] Marwa Shousha, Lionel Briand, and Yvan Labiche. A UML/MARTE model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. *IEEE Transactions on Software Engineering*, 38(2):354–374, 2012.

[176] Stephen F. Siegel. Model checking nonblocking MPI programs. In *Verification, Model Checking, and Abstract Interpretation*, pages 44–58. Springer, 2007.

[177] Stephen F. Siegel. Verifying parallel programs with MPI-Spin. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 13–14. Springer, 2007.

[178] Stephen F. Siegel and George S. Avrunin. Verification of MPI-based software for scientific computation. In *Model Checking Software*, pages 286–303. Springer, 2004.

[179] Margaret H. Smith, Gerard J. Holzmann, and Kousha Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 14–22. IEEE, 2001.

[180] Frank PM Stappers, Michel A Reniers, and Jan Friso Groote. Suitability of mCRL2 for concurrent-system design: a 2× 2 switch case study. In *Formal Methods for Components and Objects*, pages 166–185. Springer, 2010.

[181] The Eclipse Foundation. Eclipse Modeling MDT-UML2 component. URL http://www.eclipse.org/uml2.

[182] Tom A.N. Engels and others. Search algorithms for automated validation. *Journal of Logic and Algebraic Programming*, 78(4):274–287, 2009.

[183] Mirco Tribastone and Stephen Gilmore. Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile. In *Proceedings of the 7th international workshop on Software and performance*, pages 67–78. ACM, 2008.

[184] Mirco Tribastone and Stephen Gilmore. Automatic translation of UML sequence diagrams into PEPA models. In *Proceedings of the 5th International Conference on Quantitative Evaluation of Systems*, pages 205–214. IEEE, 2008.

[185] Andrei Tsaregorodtsev, Adrian Casajus Ramo, and Ricardo Graciani Diaz. DIRAC | The INTERWARE. URL http://diracgrid.org.

[186] Andrei Tsaregorodtsev et al. DIRAC: a community grid solution. *Journal of Physics: Conference Series*, 119(6):062048, 2008.

[187] Kenneth J. Turner et al. *Using formal description techniques: an introduction to Estelle, LOTOS and SDL*, volume 85. Wiley, 1993.

[188] Wil M.P. Van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets 1997*, pages 407–426. Springer, 1997.

[189] Wil M.P. Van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[190] Wil M.P. van der Aalst, H.T. De Beer, and Boudewijn F. van Dongen. Process mining and verification of properties: an approach based on temporal logic. In *Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems*, pages 130–147. Springer-Verlag, 2005.

[191] Rob J. van Glabbeek. What is branching time semantics and why to use it? *Current Trends in Theoretical Computer Science; Entering the 21st Century*, pages 469–479, 2001.

[192] Moshe Vardi. Branching vs. Linear Time: Final Showdown. In *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 1–22. Springer-Verlag, 2001.

[193] Kees Verstoep et al. Efficient large-scale model checking. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[194] Willem Visser and Peter Mehlitz. Model checking programs with Java Pathfinder. In *Proceedings of the 12th international conference on Model Checking Software*, pages 27–27. Springer-Verlag, 2005.

[195] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257. IEEE Computer Society, 2008.

[196] Jon Whittle. Transformations and Software Modeling Languages: Automating Transformations in UML. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 227–242. Springer-Verlag, 2002.

[197] Wikipedia. Model View Controller (MVC). URL http://en.wikipedia.org/wiki/Model-view-controller.

[198] Jim Woodcock et al. Formal methods: Practice and experience. *ACM Comput-*

ing Surveys (CSUR), 41(4):19, 2009.

[199] Murray Woodside. From annotated software designs (UML SPT/MARTE) to model formalisms. In *Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation*, pages 429–467. Springer-Verlag, 2007.

[200] Nianhua Yang et al. Mapping UML Activity Diagrams to Analyzable Petri Net Models. In *Proceedings of the 10th International Conference on Quality Software*, pages 369–372. IEEE, 2010.

[201] Nadia Younsi, Abdelkrim Amirat, and Ahcene Menasria. From UML 2.0 Sequence Diagrams to PROMELA code by Graph Transformation Using AToM 3. In *CEUR Workshop Proceedings*, volume 825, 2011.

[202] Paul Ziemann and Martin Gogolla. An Extension of OCL with Temporal Logic. In *Critical Systems Development with UML*, volume 2, pages 53–62, 2002.