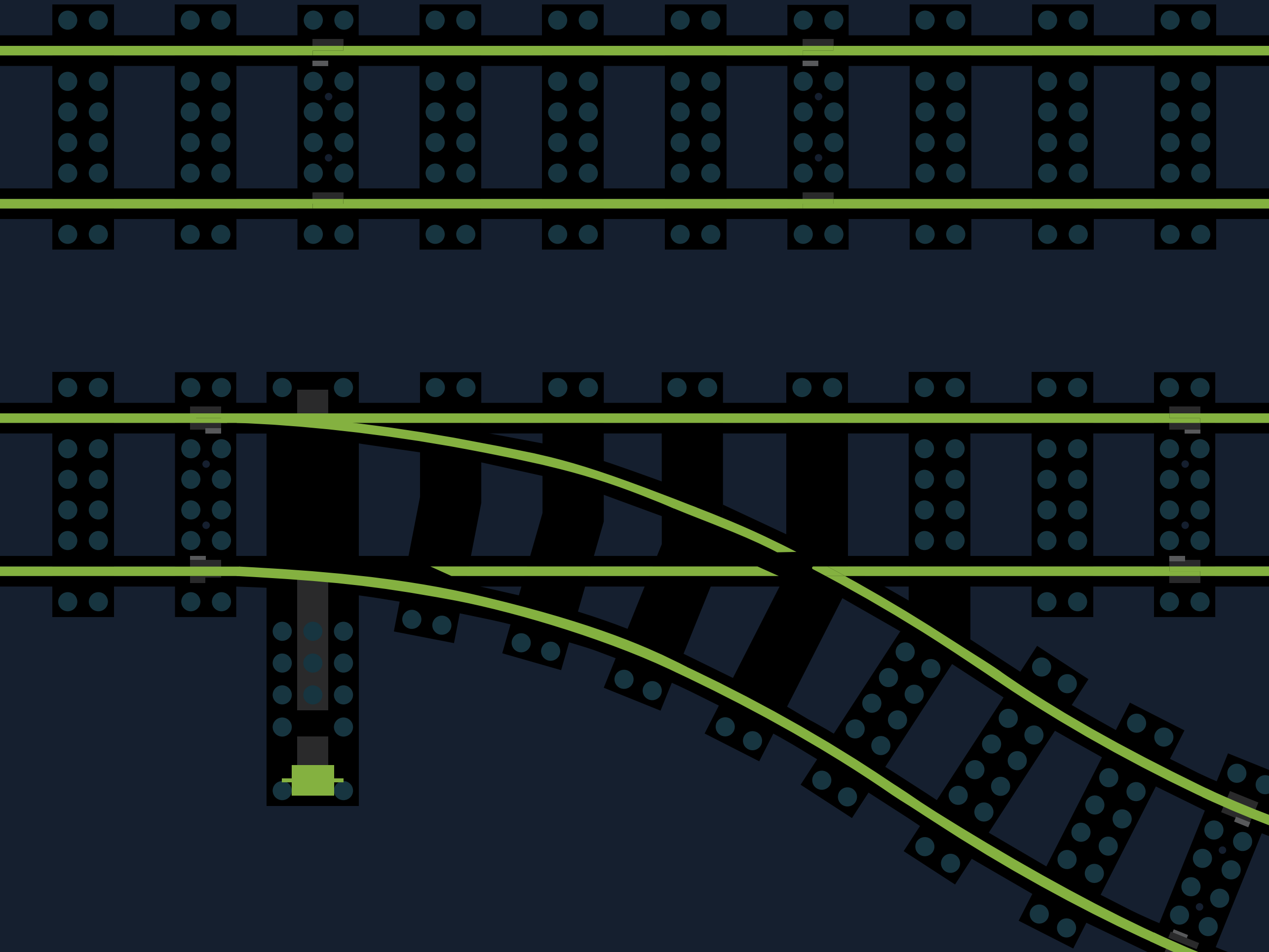
Supporting Railway Standardisation with Formal Verification

Supporting Railway Standardisation with Formal Verification

Mark Bouwman



Supporting Railway Standardisation with Formal Verification

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr. S.K. Lenaerts, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op
maandag 23 oktober 2023 om 16:00 uur

door

Markus Simeon Bouwman

geboren te Dordrecht

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2023-08.

The position of the author was fully funded by ProRail B.V. and DB Netz AG. The vision presented in this thesis does not necessarily reflect the strategy of DB Netz AG or ProRail B.V., but reflects the personal views of the author.

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-5821-6

Printed by ADC Nederland.

Cover design by Erik Hendriks.

"The only difference between screwing around
and science is writing it down."
– Adam Savage

Preface

The quote on the preceding page is from the TV show 'Mythbusters', which I loved to watch as a kid. It was my dream to one day join the Mythbusters team, for two big reasons: learning by experimenting and, of course, blowing stuff up. The adult version of this dream is to become a researcher; to 'screw around' until you discover something new, with the thrill that you might find knowledge no human has ever held before. Doing a PhD was, for me, a means to an end: to learn how to do proper research.

During most of my time as a PhD student coming up with ideas to tackle problems wasn't the most difficult part, I was already pretty good at screwing around until I found something that seemed to work. The thing I needed to learn to become a scientist was to write it down properly. Whilst writing things down I sometimes discovered it only *seemed* to work. The three big challenges I faced in converting ideas to a scientific text are i) writing proper definitions formalising the ideas, ii) proving that the goal is met with the proposed techniques, and iii) writing it down in such a way that someone else also understands. As if that wasn't difficult enough, there are usually two more challenges when trying to publish your work: i) you need to squeeze everything within a page limit and ii) you need to convince the first readers of your paper (the reviewers) that your work is novel and impactful. The reader of this thesis can judge whether my learning trajectory w.r.t. writing has been successful.

The saying "practise makes perfect" is partly true. It is true in the sense that you cannot learn new skills (such as writing) without actually doing it. However, to become better, it is helpful to gain insight in what you did well and what can be improved; feedback from a more skilled supervisor is therefore immensely valuable. In this respect I could not have wished for a better supervisor than Bas Luttik. He always gave detailed and helpful feedback when I sent him a document to review. Moreover, he always encouraged me to just make a small step forwards whenever I was paralysed by not seeing the complete path to the final solution. Thank you Bas for helping me grow professionally.

I also want to thank Rick Erkens for the fun and exciting cooperation in developing a fast rewrite engine. When I was learning the Rust programming language I was searching for a nice project to go beyond toy examples. Rick had just developed a new (and rather complicated) term matching algorithm and was reluctant to implement it, preferring to stick to the theory. He gave me the challenge of implementing his algorithm, which I managed within a couple of weeks. This first prototype was incredibly slow though. Intrigued by the elegance of the algorithm and the thrill of the need for speed I continued optimising the implementation as a hobby project. Over the course of 1.5 years we came to a tool that can compete with the state-of-the-art. Our cooperation was a match made in heaven; Rick could prove that his algorithm could be efficiently implemented and I could experiment and learn some programming whilst Rick was burdened with writing it all down and proving it correct.

Next, I would like to extend my gratitude to Djurre van der Wal, my fellow PhD student in the FormaSig project. Thank you for your major contributions in creating the SysML to mCRL2 translation framework, without you it would never have become so elegantly structured. Complements also for your (Java) programming skills. Over the years I encountered only 1 or 2 bugs in the UnifyingBlock you created, which is pretty remarkable for a prototype scientific tool.

My gratitude also goes out to Tim Willemse and Jan Friso Groote who have both been chair of the FSA group during my period as a PhD student. Tim, thank you for the fun cooperation on several topics and all the time you have invested in proofreading my manuscripts. Jan Friso, thank you for your tireless efforts to advance the mCRL2 toolset and your wonderful anecdotes on the application of formal methods in industry.

One of the benefits of conducting research at a university is that you are surrounded by others who share your interests in computer science and know the struggles of research. As it is written: "Two are better than one, because they have a good return for their labor: If either of them falls down, one can help the other up."[1]. My thanks to all the colleagues of the FSA and SET groups with whom I have exchanged ideas with, had lunch with or shared a beer with: Bas Luttik, Tim Willemse, Jan Friso Groote, Hans Zantema, Jeroen Keiren, Erik de Vink, Anton Wijs, Wieger Wesselink, Thomas Neele, Clemens Dubslaff, Roel Bloo, Rick Erkens, Olav Bunte, Maurice Laveaux, Tom Franken, Anna Stramaglia, Kevin Jilissen, Jan Martens, Ferry Timmers, Flip van Spaendonck, Isabelle Cooijmans, Nathan Cassee, Lars van den Haak, Sangeeth Kochanthara, David Manrique Negrin, Hossain Muhammad Muctadir, Lina Ochoa Venegas, Weslley Torres and Agnes van den Reek.

My gratitude also goes out to the members of the doctorate committee: Tim Willemse, Bas Luttik, Twan Basten, Jan Friso Groote, Jan Peleska, Maurice ter Beek and Susanne Lange. Thank you for all the time and effort you put into reading and judging my thesis. Your feedback has definitely helped to improve it.

Finally, I would like to thank my dear wife Nienke. You supported me throughout the journey of the past 4 years. You encouraged me and lifted up my spirits when

[1]Ecclesiastes 4:9-10, NIV

I felt like an impostor when I was stuck on a problem, and you also shared in my joy when I had a breakthrough.

Lastly, I would like to share a text that I read every time I was worrying: “Look at the birds. They don’t plant or harvest or store food in barns, for your heavenly Father feeds them. And aren’t you far more valuable to Him than they are? Can all your worries add a single moment to your life? Seek the Kingdom of God above all else, and live righteously, and He will give you everything you need. So do not worry about tomorrow; for tomorrow will worry about itself. Each day has enough trouble of its own." [2]

[2]Matthew 6:26,27,33,34, NLT and NASB

Contents

Chapter 1

Introduction

The study of truth requires a considerable effort - which is why few are willing to undertake it out of love of knowledge - despite the fact that God has implanted a natural appetite for such knowledge in the minds of men.

Thomas Aquinas, Summa Contra Gentiles

In all engineering domains the correct functioning of systems and protocols is important. In some domains though, correctness can be a matter of life and death, such as in the medical, aerospace and railway domains and in the oil and gas industry. Also when the cost of equipment is extremely high, like in high tech industry, correctness is extra important. In these domains ensuring that the system *always* works safely and correctly is perhaps an even bigger challenge than ensuring the system works at all.

We focus on the railway domain. Over the years many procedures and fail-safe designs have been developed to make the railways safer and safer. In many cases, a fatal accident led to improved safety measures [34]. After more than 200 years of rail innovation, travelling by train is very safe, much safer than travelling by car [25].

Railway safety has many aspects. For example, the tracks and trains need to be mechanically robust to prevent derailment, and animals digging tunnels near the tracks need to be monitored and removed before posing a danger. Another important aspect is railway signalling, which forms the context of this thesis. A signalling system is used to control the movement of trains, in particular to avoid collisions. Developing safe railway signalling systems is a tough engineering challenge. From the design of higher level protocols to decide when a train is allowed to move to the electro-mechanical implementation and to guidelines for train drivers: scrutiny is essential as a single flaw could potentially lead to an unacceptable hazard.

A railway signalling infrastructure consists of many *field elements* such as points, light signals, level crossings, and train detection systems. Traffic controllers are human operators setting the routes for trains. It would be unsafe to let traffic controllers directly manipulate the infrastructure, as there could be human errors. Instead there is a safety system between the traffic control systems and the trackside equipment: the *interlocking*. The interlocking controls all the field elements. The traffic control system requests a route for a train from the interlocking. If the interlocking decides that this route does not conflict with the route of any other train, it sets the route (by moving points and setting signals) and monitors the route as the train moves over it. The interlocking reports the status of the infrastructure to the traffic controllers.

Typically, the design of new railway equipment is approved by going through several rounds of human reviews with the goal of catching any potential flaws. This has the potential for human error, but for simple signalling systems this is generally sufficient to attain enough confidence that the design is correct. However, over the last few decades the railway industry is switching from relatively simple electro-mechanical systems to more advanced digital systems. These digital systems have many benefits but a downside is that specifications are getting more complex and therefore it becomes harder to get sufficient confidence in the correctness. To cope with the complexity, railway engineers are gradually adopting a Model-Based System Engineering (MBSE) approach for the development of their systems. In MBSE the system being designed is captured using a *modelling language*. The idea is that an MBSE approach ensures that specifications are more structured and therefore easier to comprehend and reason about.

There are many modelling languages with varying degrees of formality. Some modelling languages define the meaning (i.e. semantics) of language constructs using natural language. For such languages the syntax may be clear and well defined but the semantics of the modelling language may be ambiguous; this class of modelling languages is said to be semi-formal. Other languages capture the semantics of the language using mathematics and logic; such languages are said to be formal. Some formal languages come with a toolkit providing automated analysis techniques. The benefits of such tools is that they can analyse much more scenarios than a human ever could or even *prove* that the model meets certain requirements.

1.1 Formal Verification

There are many formal languages with a diverse range of purposes. We are interested in modelling languages that capture the behaviour of the system being modelled. One such modelling language is actively maintained at Eindhoven University of Technology: mCRL2 [24]. The associated toolkit[1] can reason about the behaviour

[1]`https://mcrl2.org`

of models. In fact, it can compute the *state space* of a model containing every state that can be reached from the initial state. For example, the state space induced by a model capturing the game '4 in a row' contains a state for every configuration of the board that is possible by playing the game.

The mCRL2 toolkit also allows us to check properties for a model, which is called *model checking*. For example, given the '4 in a row' model we can check the property 'Player 1 has a winning strategy' by formalising it with a logical formula and using the mCRL2 toolkit. The toolkit will then explore the state space, conclude the property does not hold and give a counterexample to the user.

Since the semantics and all the analysis techniques are firmly grounded in mathematics the result of model checking truly constitutes a *proof* that the property does (not) hold for the model. For models of safety-critical systems it is a big benefit that such a high degree of certainty can be achieved.

Formal methods have been used extensively within the railway domain [39, 40]. A recent survey [4] on the uptake of formal methods in the railway domain shows that a wide range of methods and tools are used in academia and industry, most notably the B method (and variants of it). Both mCRL2 and its predecessor μCRL [8] have been used in the signalling domain before. The μCRL toolkit has been used to assess the safety of a Vital Processor Interlocking at the station Hoorn-KersenBoogerd in the Netherlands [57]. Similarly, model checking with mCRL2 has been applied to a Siemens digital interlocking [14]. The mCRL2 toolset has also been used to verify the correctness of the ERTMS hybrid level 3 principles [3].

Many applications of formal methods focus on the interlocking, since it is the central safety system coordinating the field elements. For example, in [71] the behaviour of interlockings is specified and verified using CSP||B [112]. The formal specification language EURIS [31] was specifically developed to specify interlocking logic. In recent years there have also been a number of publications on the application of bounded model checking in the railway domain [9, 64, 65, 69, 72]. For example, in [69] a combination of bounded model checking and inductive reasoning is used to verify the correctness of a Danish interlocking system.

The field of formal methods in the railway domain is broader than verifying requirements for the state space induced by a model. For example, in [88] an approach is presented to automatically check whether a track layout (the positions of tracks, points, signals, etcetera) is compliant with national regulations. In [60] a technique called Fault Tree Analysis is used to find the most efficient way to meet RAMS (Reliability, Availability, Maintenance, Safety) requirements.

Another approach to formal verification is tool assisted theorem proving. In this approach the state space does not need to be explored by the toolset. In [63] a distributed railway interlocking is verified using the RAISE toolset [94].

Formal methods are also used in other domains, especially in the aforementioned domains where the correct and safe functioning of systems is extra important. Examples of systems that have been analysed with formal methods include control software of the Large Hadron Collider [70], satellite software [37] and aircraft software [113].

In the context of this thesis we focus on model checking signalling specifications using mCRL2. In particular, we verify the correctness of a European standard for the communication between the interlocking and the various field elements that is in development. This standard is introduced in the next section.

1.2 EULYNX

In traditional signalling setups, the motors, sensors and lights of the field elements are directly connected to the interlocking via electrical wires. This makes the implementation of a field element highly dependent on the implementation of the interlocking, reducing interoperability. Moreover, the copper wiring is expensive and a single point of failure; when a wire is cut accidentally while digging, it immediately results in a dysfunctional field element.

EULYNX[2], an initiative of a consortium of thirteen European railway infrastructure managers, aims to standardise the interfaces between the interlocking and the field elements. A key innovation by EULYNX is to eliminate the direct electrical connections between interlocking and field elements and switch to communication over an IP network. The low level hardware is controlled locally by an *object controller*, which in turn communicates with the interlocking. By standardising the interface, EULYNX creates a larger European market for field elements and decouples the life cycles of the interlocking and field elements. Moreover, EULYNX communication offers a higher resilience to cable failures, since IP packets can be routed dynamically.

EULYNX has adopted an MBSE approach to improve clarity and precision compared to the traditional natural language specifications and to enable simulation of models. EULYNX specifications are given in a dialect of SysML [97]. SysML is a popular systems engineering modelling language, closely related to the Unified Modelling Language (UML [95]). SysML defines nine different diagram types, several of which are extended versions of UML diagram types. Five are used in EULYNX specifications: block definition diagrams, internal block diagrams, use case diagrams, sequence diagrams, and state machine diagrams. Each interface with a field elements has its own SysML model, consisting of a collection of diagrams.

The use of SysML for system specifications is a big step forward for the railway domain as it is significantly more precise than natural language. SysML has a fairly intuitive graphical syntax, which allows railway engineers to understand and use it without extensive training. Still, SysML is *semi-formal*: it has a well-defined syntax, but its semantics is informal and not firmly grounded in mathematics. As a consequence, system behaviour specified by a SysML model is not directly amenable to the more thorough kind of analysis that genuine *formal* methods offer.

[2]See `https://eulynx.eu`.

1.3 FormaSig

This thesis is part of the *FormaSig*[3] project, a collaboration of the Dutch and German railway infrastructure managers, Eindhoven University of Technology and the University of Twente, with the aim to formalise the aforementioned EULYNX standard to the extent that delivered components conforming to the standard provably satisfy a collection of safety properties. The idea is to associate with each EULYNX SysML model a formal mCRL2 model [24, 55]. Then mCRL2's model checker can be used to establish that the model satisfies the required safety properties (see Figure 1.1). Additionally, test cases can be automatically derived from the model to reliably test compliance of actual implementations to the model.

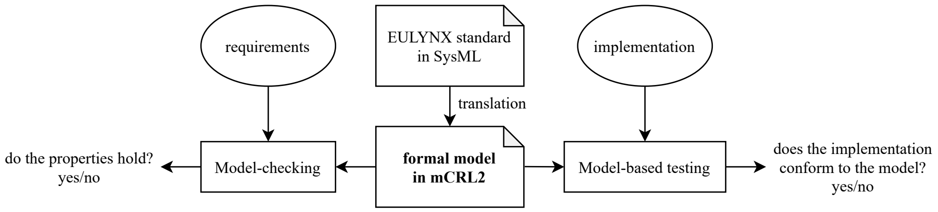


Figure 1.1: FormaSig: using a formal mCRL2 model to establish that implementations conforming to the EULYNX standard satisfy properties.

The main goal of FormaSig: supporting the development of railway standardisation by providing a formal interpretation of EULYNX models, analysing whether safety requirements are met and testing whether implementations conform to the provably correct model. Research is needed to achieve this goal. This thesis focusses on the aspects of formalisation and model checking. The main research question treated in this thesis is: *how can we formalise the semantics of SysML and effectively verify requirements for industrial models?* The aspect of model-based testing is covered by a second PhD student in the FormaSig project, Djurre van der Wal, who is employed by the University of Twente. The limited overlap in our work is detailed in the next section.

The first challenge in achieving the stated research goal is to create a way of (automatically) deriving an mCRL2 model from a SysML model consisting of multiple diagrams. To determine the complete behaviour of the EULYNX interfaces and to generate the complete mCRL2 model, internal block diagrams and state machine diagrams are sufficient. Internal block diagrams describe the interfaces of components and how they are connected. State machine diagrams specify the behaviour of components. Both types of diagrams are introduced in more detail in Chapter 3. Two questions need to be answered. How can we derive an mCRL2 model from a set of internal block diagrams and state machine diagrams? What ambiguities are present in SysML and how do we resolve them?

Another challenge is scalability. Does our formalisation lead to mCRL2 models that are amenable to model checking? If not, what techniques can we use to

[3]Formal Methods in Railway Signaling Infrastructure Standardization Processes

increase scalability?

Once we have models that are amenable to model checking we need to elicit requirements. This is challenging as railway engineers do not have experience with formulating formal requirements and we as academics lack the needed domain knowledge. How do we obtain the right requirements? Do the models satisfy these requirements? If not, how can the models be adapted such that the flaws are repaired?

SysML is a state-based modelling language whereas mCRL2 is action based. Some requirements may refer to states in the SysML model. How do we formulate such requirements exclusively in terms of actions? Is it possible to integrate the state based and action-based paradigms?

For liveness requirements it is typical that some assumptions need to be made on how events are scheduled to rule out unrealistic runs of the system that violate the requirement. EULYNX does not make any scheduling assumptions. What is the least assumption we can make in this regard that rules out sufficiently many unrealistic computations? How can we rule out unrealistic computations while model checking?

These questions are addressed in upcoming chapters and are reflected upon in the Conclusion (Chapter 8). The next section outlines the structure of the thesis and thereby what chapter addresses which question.

1.4 Contributions and Structure

The overarching narrative of this thesis is the verification of EULYNX SysML models. Some of the contributions are directly related to the main topic, whilst others are a more general contribution to the field of model checking. For these more general contributions we show how they have been or could be employed to aid in the verification of EULYNX models.

Below we examine the contributions in this thesis on a chapter by chapter basis. The next chapter, Chapter 2, introduces the preliminaries on the mCRL2 language and toolkit that are necessary to understand the rest of the thesis and does not provide any contributions.

Various models, requirements and scripts are made available through a Zenodo repository [12]. Throughout this thesis we refer to specific items that are available in the repository.

The order of the chapters in this thesis does not reflect the chronological order of the work performed in FormaSig. There were feedback loops between trying to verify requirements, adjusting the formalisation and developing scalability improvements. In the case studies described in this thesis we use the latest toolchain but also reflect on how the case studies have improved the toolchain.

Chapter 3: Formalisation. The first step in our journey to verify EULYNX SysML models, is to transform them to mCRL2 models. In this chapter we propose such a translation. The approach taken is to encode the structure of the SysML state machines in the data language of mCRL2. This specific encoding of the SysML model is combined with a generic mCRL2 model fragment defining the semantics of the state machine elements. This chapter focusses on the latter, a specification of the semantics of state machines directly in mCRL2.

This chapter is an extension of the following conference paper.

> Mark Bouwman, Bas Luttik, and Djurre van der Wal. "A Formalisation of SysML State Machines in mCRL2" [17].

Mark Bouwman has been the main contributor of the formalisation in mCRL2. Djurre van der Wal's contribution has focussed on a translation tool to generate the mCRL2 encoding from a set of SysML diagrams. His work is referred to but not included in this thesis.

Chapter 4: Scalability. A common challenge in model checking is scalability. A system typically contains multiple parallel components and the state space tends to scale exponentially with the number of components. In this chapter we explore a number of techniques to reduce the size of the state space induced by mCRL2 models obtained through our SysML translation. In particular, we discuss a technique called compositional minimisation, which turns out to be very effective for our models. With this technique the state space of the entire (monolithic) model is not computed in one go. The model is first split into components. The state space of each component is computed and then minimised modulo an equivalence relation, reducing the number of states. These minimised state spaces are finally combined to construct the state space of the entire model.

This chapter is an extension of the following conference paper.

> Mark Bouwman, Maurice Laveaux, Bas Luttik, and Tim A. C. Willemse. "Decompositional Branching Bisimulation Minimisation of Monolithic Processes" [15].

The compositional minimisation technique stems from work by Maurice Laveaux and Tim Willemse, originally only supporting strong bisimulation minimisation. This thesis and the conference paper [15] repeat the basic definitions of the earlier work but focus on an extension to branching bisimulation, of which Mark Bouwman has been the lead author. Additionally, the extended technique is applied in FormaSig yielding a much bigger reduction than is possible with just strong bisimulation minimisation.

Chapter 5: Case Studies. The preceding chapters have given us the necessary tools to verify requirements for FormaSig models. In this chapter we examine several real world EULYNX interfaces, formulate (safety) requirements and verify them using the mCRL2 toolset. Spoiler: not all requirements hold. Our verification efforts identified a few gaps in the EULYNX specifications.

This chapter is partially based on the following journal paper, detailing an analysis of the EULYNX point interface.

> Mark Bouwman, Djurre van der Wal, Bas Luttik, Mariëlle Stoelinga, and Arend Rensink. “A Case in Point: Verification and Testing of a EULYNX Interface” [21].

Both Djurre van der Wal and Mark Bouwman were lead authors of this paper, where Djurre van der Wal focussed on model-based testing and Mark focussed on formal verification. The description of the point interface and the sections on formal verification have been included in this thesis.

Chapter 6: Justness. As we will see in the case studies, we sometimes want to check requirements of the shape “if event **a** happens, then always eventually **b** happens”. It is typical that for such requirements you need some assumptions on how events are scheduled. These range from very basic assumptions to rather strong assumptions that may not be realistic. In recent years the notion of ‘justness’ [48] has been introduced. In short, it is the assumption that once an action is enabled that stems from a set of parallel components, then one (or more) of these components will eventually partake in an action. This appears to be a realistic assumption that is just strong enough for many liveness requirements. In this chapter we show how we can verify liveness requirements with a justness assumption for mCRL2 models. Moreover, we apply the theory to FormaSig models.

This chapter is an extension of the following journal paper.

> Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. “Off-the-shelf automated analysis of liveness properties for just paths” [18].

This paper is an extension of a seminar report by Mark Bouwman [10].

Chapter 7: Global Variables. Besides justness, the cases studies reveal another need: the concept of global variables. For some requirements for FormaSig models we need to specify that the system always (or never) ends up in a specific SysML state. The needed information on the SysML states is embedded in the states of the state space induced by the model. Recalling the example of ‘4 in a row’: each state in the state space represents some configuration of the board and the transitions represent moves by the players. In most model checking tools (including mCRL2) we can only refer to actions in the logic used to specify requirements. In this chapter we study a process algebra with global variables and a logic allowing references to these variables. This enables us to inspect part of the contents of a state during verification (such as the current SysML state or the state of the ‘4 in a row’ board). Moreover, in some settings, it allows for a more natural form of communication between parallel components (compared to message passing).

This chapter is an extension of the following conference paper.

Mark Bouwman, Bas Luttik, Wouter Schols, and Tim A. C. Willemse. "A process algebra with global variables" [16].

Mark Bouwman has been the lead author of this paper.

The thesis is concluded by Chapter 8, which includes a number of suggestions for future research directions.

Chapter 2

Preliminaries

Prepare your work outside; get everything ready for yourself in the field, and after that build your house.

Proverbs 24:27

In this chapter we present a number of preliminaries regarding the mCRL2 toolkit as aspects of it will be relevant in each subsequent chapter. After reading the preliminaries, the remaining chapters can be read more or less independently. We do not cover all the details of mCRL2 in detail but just the aspects that are important for this thesis. We clearly marked which aspects are left out. We generally refer to the mCRL2 book [55] for a more complete overview.

The remainder of this chapter is organised as follows. Section 2.1 provides a first impression of the mCRL2 language. Next we go over relevant process algebraic definitions (Section 2.2) and introduce the logic used by mCRL2: the modal μ-calculus (Section 2.3). Finally, in Section 2.4 we present some common workflows with the mCRL2 toolkit that are used throughout the thesis.

2.1 Introduction to mCRL2

The mCRL2 toolset [24] is designed to analyse concurrent and distributed systems. The mCRL2 language is an ACP-style [6] process algebra and contains a rich data language based on abstract data types. The semantic interpretation of an mCRL2 model is a Labelled Transition System (LTS). The toolset contains tools for the verification of parametrised modal μ-calculus formulas, bisimulation reduction, counterexample generation [30, 116], simulation and visualisation. Before presenting the formal underpinning in Section 2.2 we first informally introduce the mCRL2 language with an example.

The mCRL2 language has some primitive data types built in, such as integers, natural numbers and Booleans, including common operations on them. Users can also define their own data types and operations. The code below shows an example. The *sort* `Place` has one *constructor*, `Coordinates`, with *projection functions* `X` and `Y`. One operation is defined on `Places`. In the `map` section the operation is declared with its data signature. The `var` section declares some variables which are used in the `eqn` section to define the operation.

```
sort
  Place = struct Coordinates(X:Nat, Y: Nat);
map
  computeManhattanDistance: Place#Place -> Nat;
var
  p1, p2:Place;
eqn
  computeManhattanDistance(p1,p2) = abs(X(p1)-X(p2))+abs(Y(p1)-Y(p2));
```

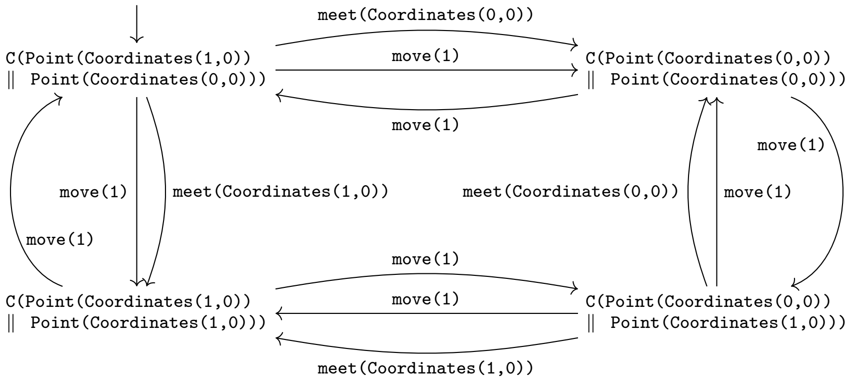
The process definition below specifies the behaviour of the `Point` process; it can perform three actions: `move`, `invite` and `respond`, which are declared in the `act` section with their type signature. Note that we also declare an action `meet`; this will be used in a moment to denote the communication result of `invite` and `respond`. The + operator represents a non-deterministic choice (not to be confused with the same symbol for addition). The sum operator represents a non-deterministic choice over a data domain. Summations over infinite data domains can be restricted by adding a guard. In the example below a guard is used to restrict the `move` action to any place, other than the current place `p`, on a 2 by 1 grid.

```
act
  move: Nat;
  invite, respond, meet: Place;
proc
  Point(p:Place) =
    sum new:Place. (new != p && X(new) < 2 && Y(new) < 1)
      -> move(computeManhattanDistance(p,new)).Point(new)
    + invite(p).Point(p)
    + sum new:Place. (new != p)  -> respond(new).Point(new);
```

The initial process expression, introduced by the keyword `init`, specifies the initial state of the transition system associated with the specification. The example below specifies a parallel composition of two `Point` processes wrapped in a communication and an allow operator. Both `Point` processes can perform a `move` action, which is allowed by the allow operator. The `invite` and `respond` actions are not allowed and hence blocked. However, the two processes can synchronize on a *multi-action* `invite|respond`, which is transformed to a `meet` action by the communication operator, which is allowed by the allow operator. The labelled transition system (sometimes referred to as the state space) associated with this specification will have exactly 4 states, representing every combination of coordinates.

```
init allow({move,meet}, comm({invite|respond -> meet},
  Point(Coordinates(1,0))||Point(Coordinates(0,0))));
```

Below we depict the LTS induced by the initial process expression. The shorthand `C(_)` abbreviates `allow({move,meet}, comm({invite|respond -> meet},_))`. The initial state is indicated by an arrow without a source state.


meet(Coordinates(0,0))
C(Point(Coordinates(1,0)) || Point(Coordinates(0,0)))
move(1)
C(Point(Coordinates(0,0)) || Point(Coordinates(0,0)))
move(1)
move(1)
move(1)
meet(Coordinates(1,0))
meet(Coordinates(0,0))
move(1)
move(1)
move(1)
C(Point(Coordinates(1,0)) || Point(Coordinates(1,0)))
move(1)
C(Point(Coordinates(0,0)) || Point(Coordinates(1,0)))
meet(Coordinates(1,0))

From the LTS we can see that in every state a transition labelled with $\mathit{move}(1)$ is possible. We could also formalise this with the following μ-calculus formula: $[\top^*]\langle \mathit{move}(1)\rangle\top$. The subformula $[\top^*]$ expresses 'after every path' and the subformula $\langle \mathit{move}(1)\rangle\top$ expresses there is an outgoing transition labelled with $\mathit{move}(1)$. The μ-calculus is introduced formally in Section 2.3.

2.2 Formal Process Algebraic Definitions

In this section we provide a more structured view of the mCRL2 language and provide some formal definitions that are relevant to the understanding of this thesis. Common process algebraic definitions, e.g. defining LTSs and (branching) bisimulation are referenced regularly later in the thesis. Regarding mCRL2 we define the structure of an mCRL2 specification and the relevant operators and their semantics, leaving out aspects such as time. The intent is to give the reader a solid understanding of mCRL2 and, in particular, its multi-action semantics.

An mCRL2 specification is subdivided into sections denoted by the keywords `sort`, `cons`, `map`, `var`, `eqn`, `act`, `proc`, `init`. The first 5 keywords are related to the data specification, which will be further explained in the next section. An action declaration section lists the actions labels and their data signature. These action labels can be used in the definition of processes, which is marked by the keyword `proc`. The process specification section lists a number of (recursive) processes. Each process has a name, a list of data parameters and a defining process expression. Lastly, the `init` section defines the initial process expression, which will also be the initial state in the LTS. With the exception of the initialisation section all sections can occur multiple times.

2.2.1 Data Specification

The mCRL2 formalism features a rich data language based on abstract data types [87] and is rooted in the field of universal algebra [26]. There are several predefined sorts, e.g. to define lists and sets. In particular, in the presentation of the theory, we rely on the existence of the sorts `Bool` and `Nat` and their associated semantic domains of Booleans ($\mathbb{B}$) and natural numbers ($\mathbb{N}$), respectively. One can declare custom data types using the `sort` section, possibly with an accompanying `cons` section declaring the constructors. Structured sorts, also called recursive sorts, are used extensively throughout this thesis. They concisely enumerate the constructors of a sort, where each constructor may contain data. The `Place` sort in Section 2.1 is a structured sort with just one constructor. Below we provide another example with two constructors. The optional recogniser functions `?isOn` and `?isOff` are functions from `Status` to the Booleans.

```
sort
  Status = struct On?isOn | Off?isOff;
```

Sections marked by `map` declare a list of functions/operators with their associated signature. It is of course not sufficient to only declare the type of an operator, we also need to specify how the operator transforms the data it is given, which is achieved by the `var` and `eqn` sections. Respectively, they declare a number of variables and a list of equations (using the declared variables). Expressions are, as usual, built from variables and function symbols, e.g. `true` is an expression and `true` $\wedge x$ is also an expression. Two closed expressions are considered equal if they can be transformed into each other using the equations. In theory the equations can be used from left to right and from right to left. In the implementation, however, the equations are treated as rewrite rules and only applied from left to right.

For the remaining formal presentation we abstract from the details and presuppose some abstract data theory. For each sort D we assume the existence of a non-empty semantic domain denoted by $\mathbb{D}$. We use $e : D$ to indicate that e is an expression of sort D. The set of free variables of an expression e is denoted $\mathsf{FV}(e)$. An expression e is *closed* if and only if $\mathsf{FV}(e) = \emptyset$. A substitution σ is a total function from variables to closed data expressions of their corresponding sort. We write $\sigma[x \leftarrow e]$ to denote the substitution σ' such that $\sigma'(x) = e$ and for all $y \neq x$, we have $\sigma'(y) = \sigma(y)$. We use $\sigma(e)$ to denote the result of applying substitution σ to expression e: for each variable x each occurrence of x in expression e is replaced by $\sigma(x)$.

We presuppose a fixed *interpretation* function, denoted by $[\![\ldots]\!]$, which maps closed syntactic objects to values within their corresponding semantic domain. Semantic objects are typeset in *boldface*, e.g., the semantics of expression $1 + 1$ is $\mathbf{2}$. We denote *data equivalence* by $e \approx f$, which is true if, and only if, $[\![e]\!] = [\![f]\!]$. Note that this presentation deviates from the mCRL2 semantics, which uses model class semantics [55, Section 15.1.5].

We denote a *vector* of length $n + 1$ by $\vec{d} = \langle d_0, \ldots, d_n\rangle$. Two vectors are equivalent, denoted by $\langle d_0, \ldots, d_n\rangle \approx \langle e_0, \ldots, e_n\rangle$, if and only if their elements are *pairwise equivalent*, i.e., $d_i \approx e_i$ for all $0 \leq i \leq n$. For a vector of data expressions with their corresponding sorts $\langle d_0 : D_0, \ldots, d_n : D_n\rangle$, we write $\vec{d} : \vec{D}$. Let $\vec{d}.i$ denote the i^{th} element of $\vec{d}$. Finally, we define $\mathsf{Vars}(\vec{d}) = \{d_0, \ldots, d_n\}$.

2.2.2 Multi-actions and Transition Systems

A *multi-set* over a set A is a set with *multiplicity* for each element in A. Formally, a multi-set is a total function $m : A \rightarrow \mathbb{N}$, where $m(a)$ is the multiplicity of a. As notation we use $\wr \ldots \int$ for a multi-set where the multiplicity of each element is either written next to it or omitted when it is one. Elements with multiplicity 0 are omitted. For instance, $\wr a : 2, b \int$ over set $\{a, b, c\}$ has elements a and b with multiplicity two and one respectively. For multi-sets m and m' over set A, we write $m \subseteq m'$ if and only if $m(a) \leq m'(a)$ for all $a \in A$. Multi-sets $m + m'$ and $m - m'$ are defined pointwise: $(m + m')(a) = m(a) + m'(a)$ and $(m - m')(a) = \max(m(a) - m'(a), 0)$ for all $a \in A$.

Let Λ be the set of *action labels* (as defined in the `act` sections). For simplification of presentation we assume every action has one data parameter. We use D_a to indicate the sort of action label $a \in \Lambda$, $\mathbb{D}_a$ denotes the semantic domain of D_a. The set of *actions* $\{a(\mathbf{e}) \mid a \in \Lambda, \mathbf{e} \in \mathbb{D}_a\}$ is denoted by Ψ. A *multi-action* is a multi-set over Ψ. The set of all multi-actions is denoted by Ω.

Definition 2.1. *A labelled transition system is a tuple $\mathcal{L} = (S, Act, \rightarrow)$ where S is a set of states, $Act \subseteq \Omega$ and $\rightarrow\ \subseteq S \times Act \times S$ is a labelled transition relation. Let $s \xrightarrow{\alpha} t$ denote that $(s, \alpha, t) \in \rightarrow$.*

Definition 2.2. Multi-action expressions *are defined as follows.*

$$\alpha \ ::= \ \tau \ \mid \ a(e) \ \mid \ \alpha|\alpha$$

Constant τ represents the empty *multi-action, $a \in \Lambda$ is an action label, and e is an expression of sort D_a. Let $\mathbb{M}$ denote the set of all multi-action expressions. The semantics of a multi-action expression α, given a substitution σ, is denoted by $[\![\alpha]\!]_\sigma$ and is a multi-action which is an element of Ω. It is defined inductively as follows: $[\![\tau]\!]_\sigma = \wr\int$, $[\![a(e)]\!]_\sigma = \wr a([\![\sigma(e)]\!])\int$ and $[\![\alpha|\beta]\!]_\sigma = [\![\alpha]\!]_\sigma + [\![\beta]\!]_\sigma$. If α is a closed expression then the substitution is typically omitted.*

Note that τ is considered an unobservable action. A τ-transition may change state and thus change what observable behaviour can be seen from future transitions but the τ-transition itself is not observable.

2.2.3 Linear Process Expressions

An mCRL2 specification is linearised before other tools can be applied. Linearisation brings the process expression into a normal form, yielding an LPS [55, 56]. Linearisation removes parallel composition and communication operators, whilst preserving bisimulation. An LPS contains the same sections as a regular mCRL2 specification with the difference that there is just one linear process which is called a *Linear Process Equation* (LPE).

We introduce the mCRL2 operators in two steps. We first introduce LPEs. In the next section we move on to communication between multiple processes.

Let PN be a set of process names.

Definition 2.3 ([83]). *An LPE is an equation of the form:*

$$P(\vec{d} : \vec{D}) = \sum_{e_0:E_0} c_0 \to \alpha_0 \,.\, P(\vec{g_0}) + \ \ldots \ + \sum_{e_n:E_n} c_n \to \alpha_n \,.\, P(\vec{g_n}) \ , \tag{2.1}$$

here $P \in PN$ is the process name, *$\vec{d}$ is a vector of variables which we refer to as* process parameters, *and each:*

- *E_i is a sort ranged over by* sum variable *e_i (where $e_i \notin \mathsf{Vars}(\vec{d})$),*
- *c_i is the* enabling condition, *a Boolean expression s.t. $\mathsf{FV}(c_i) \subseteq \mathsf{Vars}(\vec{d}) \cup \{e_i\}$,*
- *α_i is a multi-action expression τ or $a_1(f_1)|\ldots|a_n(f_n)$ such that each $a_k \in \Lambda$ and f_k is an expression of sort D_{a_k} such that $\mathsf{FV}(f_k) \subseteq \mathsf{Vars}(\vec{d}) \cup \{e_i\}$,*
- *$\vec{g_i}$ is an* update expression *of sort $\vec{D}$, satisfying $\mathsf{FV}(\vec{g_i}) \subseteq \mathsf{Vars}(\vec{d}) \cup \{e_i\}$.*

The $+$-operator denotes a non-deterministic choice, also called alternative composition. Each choice in an LPE is called a *summand.* We use $+_{i\in I}$ for a finite set of *indices* $I \subseteq \mathbb{N}$ as a generalised form of alternative composition, to distinguish it from $\sum_{d:D}$, which denotes alternative quantification. The $\sum$-operator describes a non-deterministic choice among the possible values of the bounded sum variable. We generalise the action sorts and the sum operator in LPEs; we permit ourselves to use multiple data parameters in actions $a(f_0, \ldots, f_k)$ and sum over multiple data domains $\sum_{e_0:E_0,\ldots,e_l:E_l}$, respectively.

Let P be the set of expressions $P(\vec{\iota})$, where $P \in PN$ and $\vec{\iota}$ is a vector of closed expressions of sort $\vec{D}$ (the sort of P). We assume there is a defining equation as in Equation 2.1 for every $P \in PN$. The labelled transition system induced by an LPE is then formally defined as follows.

Definition 2.4 ([83]). *To associate an LTS $(\mathsf{P}, \Omega, \to)$ to process expressions in P, we define the transition relation $\to$ as follows: for each LPE $P(\vec{d} : \vec{D}) = +_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g_i})$ and for all indices $i \in I$, closed expressions $\vec{\iota} : \vec{D}$ and substitutions σ such that for all j, $\sigma(\vec{d}.j) = \vec{\iota}.j$ there is a transition $P(\vec{\iota}) \xrightarrow{[\![\alpha_i]\!]_\sigma} P(\sigma(\vec{g_i}))$ if and only if $[\![\sigma(c_i)]\!] = \textbf{true}$.*

2.2.4 Communication Between Processes

We proceed by defining a language to express parallelism and interaction of LTSs; the operators are taken from mCRL2 [55].

Definition 2.5. *Let* Comm *be the set of all possible* communication *expressions of the form $a_0|\ldots|a_n \to c$ where $a_0, \ldots, a_n, c \in \Lambda$ are action labels. Let St be a set of constants representing states. We introduce the operators* communication, allow, hide *and* parallel composition*:*

$$S ::= \ s \ \mid \ \Gamma_C(S) \ \mid \ \nabla_A(S) \ \mid \ \tau_I(S) \ \mid \ S \parallel S \ .$$

Here, $s \in St$ is a state, $C \subseteq \mathsf{Comm}$ is a finite set of communication expressions, $A \subseteq 2^{\Lambda \to \mathbb{N}}$ is a non-empty finite set of finite multi-sets of action labels, and $I \subseteq \Lambda$ is a non-empty finite set of action labels. Let S denote the set of expressions that can be constructed by the grammar above, which is parametrised with the set of constants St.

Definition 2.5 is parametrised with a set of states. In particular, it can be parametrised with the states in an LTS associated to an LPE. Later we will also apply operators to combine LTSs of components.

Note that the mCRL2 language allows a more general kind of process specification in which the operators used in LPEs (alternative composition, sequential composition, summation, conditional expressions and recursion) and the operators of Definition 2.5 can be combined. Moreover, there are a few more operators that we will not discuss.

Definitions 2.6 and 2.7, respectively, define the communication and hiding *functions* on multi-actions. They will be used to define the semantics of the communication and hiding *operators* in Definition 2.8.

Definition 2.6 ([83])**.** *We define $\gamma_C \colon \Omega \to \Omega$, where $C \subseteq \mathsf{Comm}$, as follows:*

$$\begin{aligned}
\gamma_\emptyset(\omega) &= \omega \\
\gamma_C(\omega) &= \gamma_{C \setminus C_1}(\gamma_{C_1}(\omega)) \textit{ for } C_1 \subset C \\
\gamma_{\{a_0|\ldots|a_n \to c\}}(\omega) &= \begin{cases} \{\!\!\{ c(\mathbf{d}) \}\!\!\} + \gamma_{\{a_0|\ldots|a_n \to c\}}(\omega - \{\!\!\{ a_0(\mathbf{d}), \ldots, a_n(\mathbf{d}) \}\!\!\}) \\ \quad \textit{if } \{\!\!\{ a_0(\mathbf{d}), \ldots, a_n(\mathbf{d}) \}\!\!\} \subseteq \omega \\ \omega \quad \textit{otherwise} \end{cases}
\end{aligned}$$

So, for example, note that $\gamma_{\{a|b \to c\}}(a|d|b) = c|d$. Two restrictions are necessary for γ_C to be well-defined. We require that the left-hand sides of the communications do not share labels. For example, $\gamma_{\{a|b \to c, a|d \to c\}}(a|d|b)$ would be problematic as it could evaluate to both $b|c$ and $d|c$. Furthermore, the action label on the right-hand side must not occur in the left-hand side of any other communication. For example, $\gamma_{\{a \to c, c \to a\}}(a)$ would not be well defined because it could evaluate to both a and c.

Definition 2.7 ([83])**.** *Let $\omega \in \Omega$ and $I \subseteq \Lambda$. We define $\theta_I(\omega)$ as the multi-set ω' defined as:*

$$\omega'(a(\mathbf{d})) = \begin{cases} 0 & \textit{if } a \in I \\ \omega(a(\mathbf{d})) & \textit{otherwise} \end{cases}$$

Given a multi-action expression α we write $\underline{\alpha}$ to denote the multi-set of action labels that occur in α, i.e. we remove the data expressions from the multi-action expression. Formally, $\underline{a(e)} = \{\!\!\{ a \}\!\!\}$, $\underline{\tau} = \{\!\!\{\}\!\!\}$ and $\underline{\alpha|\beta} = \underline{\alpha} + \underline{\beta}$. We define $\underline{\omega}$ for $\omega \in \Omega$ in a similar way.

Definition 2.8. *Let $(St, Act, \to_1)$ be an LTS. We associate an LTS $(\mathsf{S}, \Omega, \to)$ to expressions of S, where S is parametrised with constants St. The relation $\to$ is the* *least relation including the transition relation $\rightarrow_1$ and the rules in the Structural Operational Semantics (SOS) below. For any $\omega, \omega' \in \Omega$, expressions P, P', Q, Q' of* S *and sets $C \subseteq \mathsf{Comm}$, $A \subseteq 2^{\Lambda \rightarrow \mathbb{N}}$ and $I \subseteq \Lambda$:*

$$\text{(Par)}\ \frac{P \xrightarrow{\omega} P' \quad Q \xrightarrow{\omega'} Q'}{P \parallel Q \xrightarrow{\omega+\omega'} P' \parallel Q'} \qquad \text{(Hide)}\ \frac{P \xrightarrow{\omega} P'}{\tau_I(P) \xrightarrow{\theta_I(\omega)} \tau_I(P')}$$

$$\text{(Par-l)}\ \frac{P \xrightarrow{\omega} P'}{P \parallel Q \xrightarrow{\omega} P' \parallel Q} \qquad \text{(Par-r)}\ \frac{Q \xrightarrow{\omega} Q'}{P \parallel Q \xrightarrow{\omega} P \parallel Q'}$$

$$\text{(Comm)}\ \frac{P \xrightarrow{\omega} P'}{\Gamma_C(P) \xrightarrow{\gamma_C(\omega)} \Gamma_C(P')}$$

$$\text{(Allow)}\ \frac{P \xrightarrow{\omega} P'}{\nabla_A(P) \xrightarrow{\omega} \nabla_A(P')}\ \underline{\omega} \in A \cup \{\lbag\rbag\}$$

2.2.5 Bisimulation

We define two well-known equivalence relations on LTSs: strong bisimulation [100] and branching bisimulation [52]. They allow us to reason about the equivalence of LTSs. Moreover, the mCRL2 toolset contains a tool to *minimise* an LTS modulo an equivalence relation.

Definition 2.9. *Let $(S, Act, \rightarrow)$ be an LTS. We call a relation $\mathcal{R} \subseteq S \times S$, a* strong bisimulation relation *if, and only if, it is symmetric and for all $(s,t) \in \mathcal{R}$ the following condition holds: if $s \xrightarrow{\alpha} s'$, then there exists a state t' such that $t \xrightarrow{\alpha} t'$ and $(s', t') \in \mathcal{R}$. Two states s and t are strongly bisimilar, denoted by $s \leftrightarroweq t$, if and only if, a strong bisimulation relation $\mathcal{R}$ exists such that $(s,t) \in \mathcal{R}$.*

Let $\rightarrow^*$ denote the reflexive transitive closure of $\rightarrow$, i.e. $s \rightarrow^* t$ if and only if t is *reachable* from s. Similarly, let $\twoheadrightarrow$ denote the reflexive transitive closure of the binary relation $\xrightarrow{\lbag\rbag}$. Let $s \xrightarrow{(\omega)} t$ be an abbreviation of $s \xrightarrow{\omega} t \vee (\omega = \lbag\rbag \wedge s = t)$.

Definition 2.10. *Let $(S, Act, \rightarrow)$ be an LTS. We call a relation $\mathcal{R} \subseteq S \times S$, a* branching bisimulation relation *if and only if it is symmetric and for all $(s,t) \in \mathcal{R}$, the following condition holds: if $s \xrightarrow{\alpha} s'$ then there are states t' and t'' such that $t \twoheadrightarrow t' \xrightarrow{(\alpha)} t''$ and $(s, t') \in \mathcal{R}$ and $(s', t'') \in \mathcal{R}$.*

Two states s and t are branching bisimilar, denoted by $s \leftrightarroweq_b t$, if and only if, a branching bisimulation relation $\mathcal{R}$ exists such that $(s,t) \in \mathcal{R}$.

2.3 The Modal μ-calculus

There are several logics that can be used to specify requirements. Examples include Hennessy-Milner Logic (HML) [66], Linear Time Logic (LTL) [105], Computation Tree Logic (CTL) [28], CTL* [36] and the modal μ-calculus (L_μ). The modal

μ-calculus subsumes the expressiveness of the other mentioned logics [29]. The mCRL2 toolkit supports verification of L_μ formulas.

The modal μ-calculus can be viewed as a fixed point extension of HML. In HML one can characterise the capabilities of a state to execute actions using modal operators $[\_]\_$ and $\langle\_\rangle\_$. Essentially, this permits to reason about the transitions emanating from a state.

Example 2.11. *The HML formula $\langle a\rangle\mathsf{true}$ expresses that some outgoing a-labelled transition exists. The formula $[b]\langle a\rangle\mathsf{true}$ expresses that for every outgoing b-labelled transition we end up in a state with an outgoing a-labelled transition.*

Fixed points add the power of recursion to these basic facilities. Intuitively, this allows us to reason about finite or infinite sequences or trees of transitions and the capabilities of the states visited along such sequences or trees. The resulting logic, i.e., HML with fixed points, is referred to as the modal μ-calculus. For an in-depth treatment of this logic, we refer to, e.g., [75].

Our syntax of the modal μ-calculus is given in the context of a set of recursion variables $\mathcal{V}$. The set Φ of formulas of L_μ is generated by the following grammar (with X ranging over a set of variables $\mathcal{V}$, and λ ranging over the set of multi-actions Ω):

$$\varphi \ ::= \ X \mid \top \mid \bot \mid \varphi\wedge\varphi \mid \varphi\vee\varphi \mid \langle\lambda\rangle\varphi \mid [\lambda]\varphi \mid \mu X.\,\varphi \mid \nu X.\,\varphi \ .$$

The binding precedence of the operators is as usual: the box and diamond operators bind strongest, followed by $\wedge$, $\vee$ and finally the fixed point operators, which bind weakest. The least fixed point operator is denoted by μ and the greatest fixed point operator by ν. We permit ourselves to write $\bigwedge\limits_{\lambda\in A}\phi(\lambda)$ and $\bigvee\limits_{\lambda\in A}\phi(\lambda)$ for a set of actions $A\subseteq\Omega$, as generalisations of the binary conjunction and disjunction.

Definition 2.12. *Let $(St, Act, \rightarrow)$ be an LTS. We proceed to give a denotational semantics for our logic by associating with every formula φ the subset $[\![\varphi]\!]_\vartheta\subseteq St$ of states in which φ holds. Since formulas may contain free variables, $[\![\varphi]\!]_\vartheta$ is relative to an assignment $\vartheta:\mathcal{V}\rightarrow 2^{St}$ that provides an interpretation of recursion variables $X\in\mathcal{V}$ as subsets of St. We define $[\![\cdot]\!]_\vartheta$ recursively as follows:*

$$\begin{aligned}
[\![X]\!]_\vartheta &= \vartheta(X);\\
[\![\top]\!]_\vartheta &= St;\\
[\![\bot]\!]_\vartheta &= \emptyset;\\
[\![\varphi\wedge\psi]\!]_\vartheta &= [\![\varphi]\!]_\vartheta\cap[\![\psi]\!]_\vartheta;\\
[\![\varphi\vee\psi]\!]_\vartheta &= [\![\varphi]\!]_\vartheta\cup[\![\psi]\!]_\vartheta;\\
[\![\langle\lambda\rangle\varphi]\!]_\vartheta &= \{s\in St \mid \exists t\in St \text{ such that } s\xrightarrow{\lambda}t \text{ and } t\in[\![\varphi]\!]_\vartheta\};\\
[\![[\lambda]\varphi]\!]_\vartheta &= \{s\in St \mid \forall t\in St \text{ such that } s\xrightarrow{\lambda}t.\ t\in[\![\varphi]\!]_\vartheta\};\\
[\![\mu X.\,\varphi]\!]_\vartheta &= \bigcap\{\mathcal{F}\subseteq St \mid [\![\varphi]\!]_{\vartheta[X:=\mathcal{F}]}\subseteq\mathcal{F})\};\ \text{and}\\
[\![\nu X.\,\varphi]\!]_\vartheta &= \bigcup\{\mathcal{F}\subseteq St \mid \mathcal{F}\subseteq[\![\varphi]\!]_{\vartheta[X:=\mathcal{F}]})\}.
\end{aligned}$$

The interpretation of a formula φ is *independent* of the valuation ϑ in case it contains no unbound recursion variables (i.e., all occurrences of a recursion variable are within the scope of a least or greatest fixed point). We simply write $[\![\varphi]\!]$ when referring to the semantics of such a formula, as it yields the same set of states for every possible environment ϑ used to interpret φ.

Example 2.13. *Greatest fixed point formulas typically characterise invariant properties, whereas least fixed point formulas characterise liveness properties. For instance, the L_μ formula $\nu X.\, \langle a \rangle X \wedge [b]\bot$, asserts the existence of an infinite a-path along which no b-action can be executed. This is an invariant property along the path. On the other hand, the formula $\mu X.\, \langle a \rangle X \vee \langle b \rangle \top$ asserts that there is a finite path of a-labelled transitions, leading to a state in which a b-labelled transition is enabled.*

The logic used by mCRL2 is an extension of the μ-calculus. It allows us to use action formulas and regular formulas in the box and diamond operator, e.g. in the formula $\langle \top^*.a \rangle \bot$, $\top$ matches any multi-action and the star indicates a sequence of zero or more actions so the formula expresses that there does not exist a trace ending in an a-labelled transition. Additionally, mCRL2 allows for data parameters in the fixed points, e.g. $\mu X(n := 0).\, \langle a \rangle X(n+1) \vee (n < 5 \wedge \langle b \rangle \top)$, states that there is an a-path of length at most 4 leading to a state in which a b-labelled transition is enabled. These extensions are described in the mCRL2 book [55]. We do not describe them here formally as it would complicate matters and we only require the formal definition of the modal μ-calculus without action formulas and data parameters in later chapters.

2.4 Tools and Workflows in mCRL2

The toolset associated to the mCRL2 language offers a wide variety of tools for analysis and verification. To aid the reader in understanding upcoming chapters we discuss the tools that are used in our verification workflow. For up to date information on the mCRL2 toolset we refer to the website[1]. Figure 2.1 provides a visual representation of the workflow.

Any workflow with an mCRL2 model starts with the tool `mcrl22lps`, which linearises the model. Computing the LPS typically only takes a few seconds.

To verify a modal μ-calculus formula, a *Parametrised Boolean Equation System* or PBES [58] needs to be constructed and solved. There are two ways to construct a PBES:

- From the LTS. From the LPS we can compute the LTS associated to the LPS using `lps2lts`. The LTS can optionally be minimised w.r.t. a behavioural equivalence (e.g. branching bisimulation). Using `lts2pbes` we can construct the PBES from the LTS and a μ-calculus formula.

[1] `www.mcrl2.org`

- From the LPS. The tool `lps2pbes` generates a PBES from a μ-calculus formula and the LPS. Solving the PBES will indirectly explore (part of) the state space.

There are also two ways to solve a PBES:

- Using `pbessolve`, which provides a verdict (true/false) and, optionally, an evidence file (dis)proving the formula. The evidence contains the minimal fragment of the LTS (dis)proving the formula. The evidence file can also contain the entire state space.
- Using `pbessolvesymbolic`, which only provides a verdict. This tool represents states symbolically instead of explicitly, which is typically much more efficient. It is only effective on PBESs generated from an LPS.

In some instances the state space associated to a model is too large to explore with the explicit tools. If we still want to compute how large the state space is, we can use the tool `lpsreach`, which uses the same symbolic techniques as `pbessolvesymbolic`.

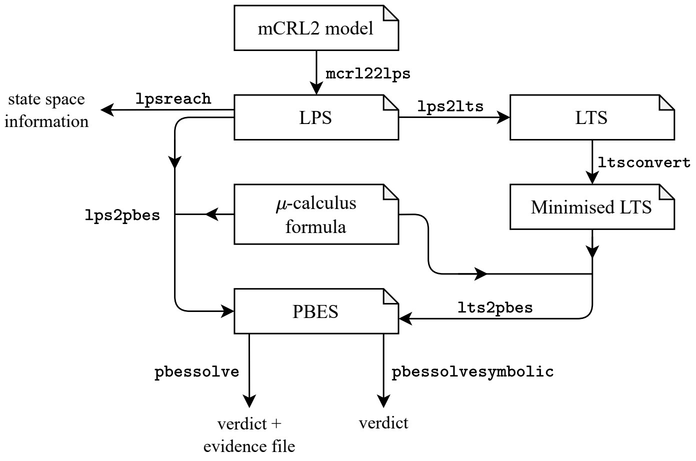


Figure 2.1: Main workflows for requirement verification in mCRL2.

Chapter 3

A Formalisation of SysML State Machines in mCRL2

Then you will know the truth, and the truth will set you free.

John 8:32

This chapter reports on our formalisation of the semi-formal modelling language SysML in the formal language mCRL2. The goal of our formalisation is to unlock formal verification and model-based testing for SysML models. The formalisation focuses on a fragment of SysML used in the railway standardisation project EULYNX. It comprises the semantics of state machines, communication between components via ports, and an action language called Atego Structured Action Language (ASAL). It turns out that the generic execution model of SysML state machines can be elegantly specified using the rich data and process languages of mCRL2. The generic model can be configured with a formal description of a specific set of state machines in a straightforward manner.

The EULYNX standard is under development, and it is likely that also in the future it will be subject to changes. Moreover, EULYNX models are large and consist of many diagrams. Hence, it is impractical to rely on manual translations from the EULYNX SysML models to mCRL2. To facilitate that model-checking and model-based test techniques will become an integral aspect of maintaining the EULYNX standard, it is imperative that the translation from EULYNX SysML to mCRL2 is automated. Another benefit of having an automated translation is that, once the translation itself is debugged and has proven itself in use, no human errors are introduced in the formalisation.

The modelling and testing cluster of EULYNX has developed a modelling standard [38] to achieve a consistent modelling style. Additionally it (informally) defines the dialect-specific constructs used in EULYNX models such as ASAL, which are linked to the PTC Windchill tool[1]. SysML contains ambiguities, which are mostly left unresolved by the EULYNX modelling standard. Consequently, FormaSig delivers *an* interpretation of EULYNX SysML. FormaSig points out ambiguities to the EULYNX modelling cluster and provides suggestions on how to resolve them, with the hope that the semantics of our formalisation and the semantics defined in the modelling standard converge.

Implementing an automated translation from SysML to mCRL2 is a non-trivial undertaking, most notably hampered by the lack of a complete and comprehensive formal semantics for SysML and the complexity of the informally described SysML execution model. Furthermore, also due to the lack of a fixed formal semantics, there are many dialects of SysML. A particular variation point is the action language, the language used to specify guards and the effects of transitions. In EULYNX SysML all communication is performed via ports, which are referenced as variables in the action language. The action language itself is ASAL.

Our formalisation consists of three parts. The first part is a generic, comprehensive formalisation of the operational semantics of UML state machines, which form the basis of EULYNX SysML state machines. This part involves formalising the notion of state hierarchy and transition selection. The second part adds an interpretation of the particular communication mechanism via ports that is used in EULYNX SysML. The third part defines an execution model for the ASAL action language. In this chapter, we generalise to a class of action languages that reference ports as variables. The resulting mCRL2 specification can straightforwardly be turned into an actual formal model interpreting a particular EULYNX SysML interface by populating the relevant data types with some static details from the SysML model and generating a suitable number of instantiations of predefined processes. We have implemented a tool to do this. The resulting mCRL2 specification can be used for model checking and model-based testing and serves as the formal model central to the FormaSig approach (see Figure 1.1).

The semantics of UML/SysML state machines has been formalised in prior academic work. A number of papers describe a translation from UML state machines to PROMELA (the input language of the SPIN model checker [68]) [80, 84, 85, 99, 111]. Our formalisation of transition selection draws inspiration from [85]. In [86] a structural operational semantics is presented along with a custom verification tool USM^{2}C. The AVATAR [101] tool offers a SysML-style environment with particular focus on verifying security properties. It offers a translation to UPPAAL [5] and ProVerif [7]. Other translations and formalisations include a translation from xUML class diagrams and state machines to mCRL2 [61, 62], a translation from SysML Block Definition Diagrams and state machines to NuSMV [115], a formalisation of UML state machines using structured graph transformations [76] and a formalisation of UML state machines in Object-Z [74]. In [67] a model-to-model transformation on Block Definition Diagrams and state machines into a single notation is proposed; from this unified model a transition relation can be derived. In [102, 103] a formal semantics is directly associated with

[1]`https://www.ptc.com/en/products/windchill/integrity/`

SysML diagrams and used for model-based testing. In [107] a translation is given from sequence diagrams to mCRL2.

Our approach to formalisation differs from earlier work in how the formalisation is achieved. In other approaches, the semantics is given by the transformation rules, or the formalisation and transformation to a formal language are distinct. In our approach, we specify the generic semantics in the target formal language itself (mCRL2) resulting in a partial model which can be instantiated with a specific configuration using the data language. Another difference is that our formalisation includes a communication mechanism using ports.

The OMG organisation, which manages the UML and SysML standards, has also released “Precise Semantics of UML State Machines (PSSM)” [98], which gives an informal but very precise semantics. Our formalisation differs in at least one way from PSSM. According to the PSSM standard a completion event is generated when a state is ‘completed’, for composite states this is the case when all parallel regions have reached the final state and for a simple state it is considered completed when its entry behaviour has finished executing. We do not create such completion events in order to prevent cluttering the event queue. Instead, transitions relying on a completion event have completion of the source state as an extra guard. PSSM also provides an extensive compliance test suite. Executing this test suite on our models would require an extensive adapter, which we have not built. PSSM only specifies the semantics of a single state machine, which does not solve ambiguities related to the concurrent execution of communicating state machines.

In Section 3.1 we give a rough sketch of the syntax of Internal Block Diagrams (IBDs) and state machines as well as the execution semantics of state machines. Section 3.2 details the formalisation strategy. Sections 3.3 to 3.7 treat the semantics of UML state machines and its formalisation in mCRL2, including the role of the action language and some mCRL2 snippets that are illustrative of the formalisation. The model itself is available on Zenodo [12]. In Section 3.8 we extend the UML semantics with EULYNX SysML specific communication over ports. In Section 3.9 we detail how to instantiate the model with a configuration. Section 3.10 discusses some adaptions for the specifics of EULYNX semantics. In Section 3.11 we present some conclusions.

3.1 Introduction to SysML IBDs and State Machines

In this section we cover the basics of the two SysML diagram types that are relevant for our formalisation. Additionally, we present some features that are specific to EULYNX.

3.1.1 Internal Block Diagrams

IBDs are used to describe the *ports* of EULYNX components. Ports are essentially variables, with a name and a type. Ports are defined as either an *input port*, denoted by an arrow-in-a-box pointing *into* the block, or as an *output port*, denoted by an arrow-in-a-box that points outwards. A component cannot change the value of its input ports.

Figure 3.1 presents the IBD of the F_SCI_P_SR component, which intermediates between the interlocking and point hardware. Its header provides the name of the component, and the area below the header lists input ports (‘T1_Cd_Move_-Point’, for example) and output ports (such as ‘T2_Msg_Point_Position’).

The IBD in Figure 3.1 declares ten *pulse ports*: four input pulse ports, recognizable by their ‘PulsedIn’ type, and six output pulse ports, recognizable by their ‘PulsedOut’ type. Pulse ports are specific to PTC/EULYNX. Their name is always prefixed by a ‘T’ and they can have the value ‘TRUE’ or ‘FALSE’, just like Boolean ports, but they automatically reset themselves from ‘TRUE’ to ‘FALSE’. The idea is that the receiver can act on the change from ‘FALSE’ to ‘TRUE’, i.e. it simulates message passing. They are frequently accompanied by *data trigger ports*, of which the names are always prefixed with ‘DT’ and has the same number as the corresponding pulse port. For example, when the T1_Cd_Move_Point pulse port of F_SCI_P_SR becomes ‘TRUE’, it means that the value of the DT1_Move_Point_Target port is (temporarily) valid and can be used to choose new behaviour. Data trigger ports add parameters to the messages. EULYNX is in the process of replacing these pulse and data trigger ports with *signals*, a standard SysML construct to model message passing. In this thesis we will not consider signals.

IBDs can also be used to define how EULYNX components are interconnected. For an example, see Figure 3.2. In such an IBD, the main block of the IBD does not define an individual component, but rather a context in which the ports of multiple components are connected by *(data) flows*. Two ports can only be connected to each other by a flow if they have the same type and opposite directions. An output port may be connected to multiple input ports, if they have the same type; input ports, on the other hand, may be connected to at most one output port. Input ports take on the data value of the output port to which they are connected.

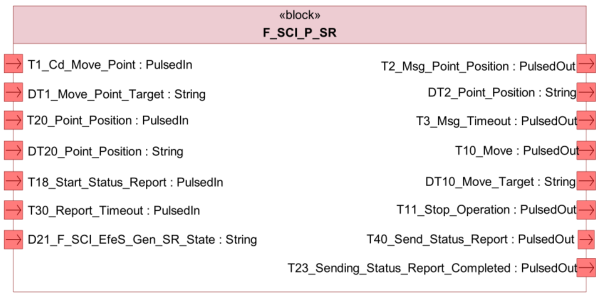

«block»
F_SCI_P_SR
T1_Cd_Move_Point : PulsedIn
DT1_Move_Point_Target : String
T20_Point_Position : PulsedIn
DT20_Point_Position : String
T18_Start_Status_Report : PulsedIn
T30_Report_Timeout : PulsedIn
D21_F_SCI_EfeS_Gen_SR_State : String
T2_Msg_Point_Position : PulsedOut
DT2_Point_Position : String
T3_Msg_Timeout : PulsedOut
T10_Move : PulsedOut
DT10_Move_Target : String
T11_Stop_Operation : PulsedOut
T40_Send_Status_Report : PulsedOut
T23_Sending_Status_Report_Completed : PulsedOut

Figure 3.1: IBD that defines the ports of the ‘F_SCI_P_SR’ component of the EULYNX Point interface. Ports have a name, a data type, and a direction (‘in’ or ‘out’).

We go over the elements of Figure 3.2. As stated in the header, the diagram defines the context called ‘SCI-P PDI SR’. The area below the header contains the components that are included in this context; among them is the component from Figure 3.1, F_SCI_P_SR, which is connected to the component S_SCI_P_SR with five flows. F_SCI_P_SR and S_SCI_P_SR have flows that lead to two other components, as well as to two objects on the edge of the diagram, labelled ‘SAP_SubS_EIL’ and ‘SAP_SubS_P’; these are *interface flow ports*, and can be used to combine contexts.

If a port of a component is not connected to another port (anywhere in the specification), it is connected to the *environment*. For an input port, this implies that its value can change non-deterministically at any moment. Supposing that the IBD in Figure 3.2 is the only IBD of a specification, the T20_Point_Position port of the F_SCI_P_SR component is manipulated by the environment. This does not hold for the two T1_Cd_Move_Point ports, because they are connected to each other.

3.1.2 State Machine Diagrams

Figure 3.3 shows an example of a state machine, with names of the various constructions added in blue. In this section, we briefly discuss the informal semantics of each construction as it is described in the UML standard [95]. In Chapter 3 we explore the semantics in more depth.

The basic constituents of state machines are *states* and *transitions*. Initial states, choice states, final states, junctions, forks and joins are also called *pseudostates*. The UML state machine formalism derives its expressiveness from these pseudostates,

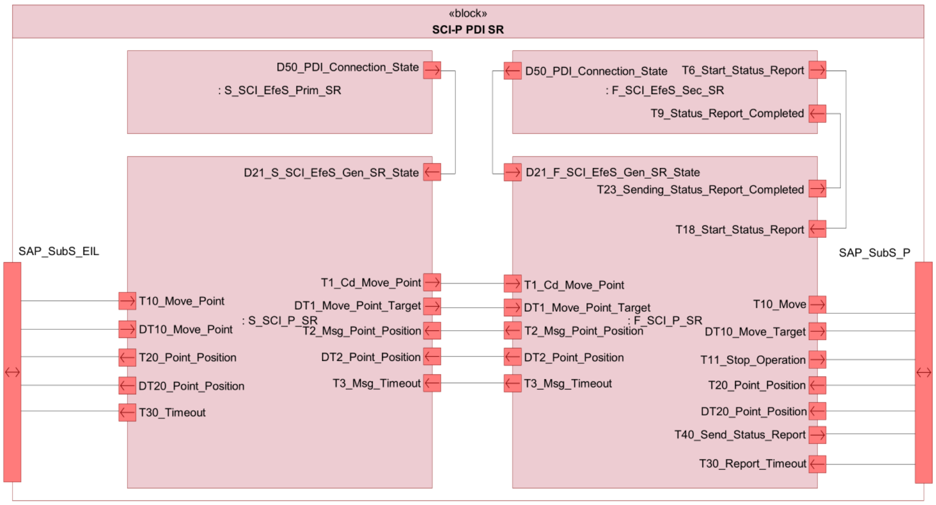


Figure 3.2: IBD that defines how certain Point components are interconnected; a so-called “context”.

the possibility to have states and transitions nested within states and the possibility to have transitions even cross state borders. Transitions may have a *trigger*, a *guard* and an *effect*. The trigger of a transition (which is optional) is an *event*; it can be a change event (notation when(x)) or a timeout event (notation after(x)).

The modeller can define behaviour that is executed upon entering or exiting a simple or composite state. *Exit behaviour* is executed when a transition from the state is fired and before the effect of that transition. *Entry behaviour* is executed after the effect of a transition by which the state is entered. Simple and composite states can also have *internal transitions*, which represent activities triggered by events that do not change state (see, e.g., the state Failed in Figure 3.3).

Junctions and *choice* vertices allow more concise specification of transitions that induce the same behaviour. The choice vertex c1 in Figure 3.3 combines two transitions from Checking which share the common behaviour A:=1. Junctions serve a similar purpose (see junction j1 in Figure 3.3). The difference between junctions and choice vertices is that for junctions the guards of outgoing transitions need to be checked before taking a transition to the junction, whereas for choice vertices the guards are checked when arriving at the vertex.

A state can contain other states, in which case it is called a *composite state* and the states it encloses are called *substates*. A composite state can have a *final state* (see, e.g., the state Failed in Figure 3.3). Transitions from the border of a composite state can be fired regardless of the current substate, except when the transition does not have a trigger, in which case the current substate of the composite state must be a final state. A composite state may also have multiple *parallel regions*. Each region has an *initial state* and can perform local transitions independently of other regions. A transition ending at the border of a composite state with parallel regions will let each region start from its initial state. A fork

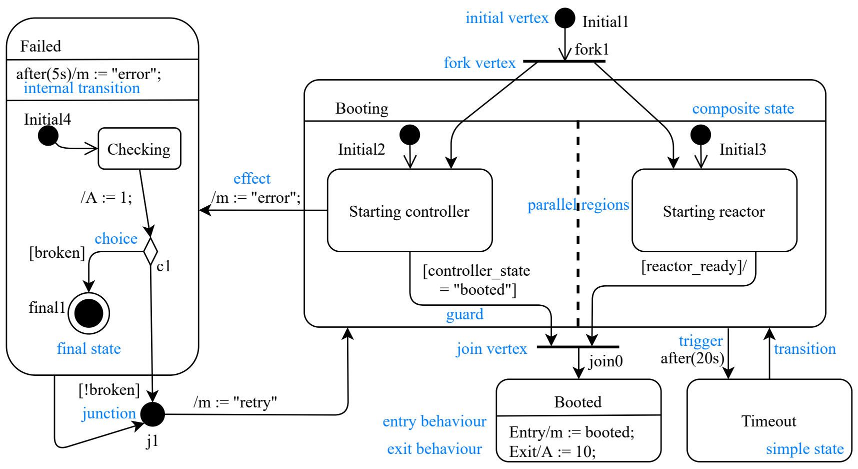


Figure 3.3: Example showing all state machine constructs supported in EULYNX SysML.

indicates that a transition ends on specific states in multiple regions. Conversely, a join can begin from specific states in parallel regions.

Due to the presence of composite states, a state machine is not just in a single state but in a collection of states, a *state configuration.* A state configuration is *stable* when it does not contain pseudostates. Transitions have a single beginning and end state. A state machine may combine several transitions (as is the case with joins, forks and junctions) to perform a bigger step from one state configuration to another, which we will call a *step.* Events that occur are stored in an *event pool* until they are either used to trigger a transition or discarded. A step is enabled when the specified trigger (if any) is in the event pool and all guards of transitions involved in the step evaluate to true. State machines have *run-to-completion* semantics: a state machine selects a step to execute and will completely finish executing the behaviour of the step and entry and exit behaviour before it considers performing a new step. Parallel regions may start a step simultaneously when both steps have the same trigger; in that case the state machine performs a *multi-step.* The behaviour of the steps in the multi-step may interleave.

3.2 Strategy to Formalisation

There are two common approaches for building a translation from an informal language to a formal language:

1. Formalise the syntax and semantics of the informal language separate from the target language (e.g. using operational rules, logic and set theory) and then define a translation to the target formalism which preserves the given semantics.

2. Define a translation to the target formalism and let the translation together with the semantic interpretation of the formal language be the semantics of the informal language. It can vary how close the representation in the target language is compared to the representation in the source language.

We have opted to use the latter strategy. We define the structure semantics directly of state machines in mCRL2, resulting in a generic model that can be instantiated for a specific SysML model. The semantic interpretation of IBDs is indirectly defined in a transformation to mCRL2. A benefit of our approach is that it provides a large degree of flexibility. It allows for an iterative process in which the semantic interpretation is adjusted, for example to improve scalability or to align with the interpretation of the EULYNX consortium.

The data specification of the generic model for state machines defines mCRL2 sorts for all the structural elements of state machines and defines operations on them to define when a transition is enabled, what behaviour is executed along a transition, etcetera. A generic StateMachine process expression defines the dynamic behaviour of a state machine. The generic mCRL2 model is not complete. A full model is created by combining the generic model with a specific configuration. The configuration consists of three parts:

- An encoding of the structure of a specific set of state machines in the mCRL2 data types defined in the generic model;
- An initial process expression spawning the correct number of StateMachine processes and passing the structure of the state machines as a parameter;
- Processes to enforce correct communication between the ports of state machines. The StateMachine processes are agnostic to how the ports are connected. It is explained later how correct communication is enforced.

Our approach has brought us a high degree of modularity. There are several choices to be made w.r.t. the semantics (e.g., w.r.t. the granularity of interleaving, run-to-completion semantics, syntax and semantics of the action language) and we want to set up our specification in such a way that parts of it can be easily modified or replaced. A particular concern is that the specific details of a concrete state machine to be translated are separated from the generic semantics.

The mCRL2 model specifying the semantics of SysML state machines is available in the Zenodo repository accompanying this thesis [12].

3.3 Abstract Action Language

The UML standard [95] is not prescriptive of the action language used to specify guards and the effect of transitions. In the spirit of modularity we have designed the generic mCRL2 encoding in such a way that it can be easily configured with a specific action language. EULYNX uses the Atego Structured Action Language (ASAL) [106]. In this section we will explain the setup in the generic mCRL2 model and how it can be configured with a specific action language.

Let `Instruction` be an mCRL2 sort containing all action language expressions. Let `VarName` be a sort containing all variable names. It is assumed that there is a single data domain for variables: the sort `Value`.

In order to formalise the action language semantics it may be necessary to include additional data structures, e.g., a program stack or a valuation of local variables. To encapsulate such additional data structures we introduce the notion of *execution frame*, represented by the mCRL2 sort `ExcFrame`, which is assumed to consist of all data necessary to execute programs of the action language. We do not assume that execution of behaviour is atomic. We allow that two components interleave their execution of behaviour when they are both taking a transition. We abstract from the granularity of interleaving and simply allow an execution frame e to make a step to an execution frame e', where e' may still have behaviour waiting to be executed.

To define the semantics of the action language, mCRL2 equations need to be added for the following mappings. We assume a subset of action language expressions represent predicates, which can be evaluated using `checkPredicate`. We assume the existence of a sort `ValueStorage` which stores the value associated with each variable.

```
sort
  VarValuePair = struct VarValuePair(getVariable:VarName, getValue:Value);
  Instructions = List(Instruction);
map
  initializeExcFrame: Instructions#ValueStorage -> ExcFrame;
  executeExcFrameCode: ExcFrame -> ExcFrame;
  checkPredicate: Instructions#ValueStorage -> Bool;
  isFinished: ExcFrame -> Bool;
  globalValuation: ExcFrame -> ValueStorage;
  varUpdates: ExcFrame -> List(VarValuePair);
  resetVariableUpdates: ExcFrame -> ExcFrame;
  getValue: ValueStorage#VarName -> Value;
  setValue: ValueStorage#VarName#Value -> ValueStorage;
```

The mapping `varUpdates` is assumed to retrieve all updates to variables that occurred during the execution of the execution frame. This field is needed for deriving change events, described at the end of Section 3.6.

ASAL has several data types: Boolean, Pulse, Integer and String. They are modelled in the mCRL2 model with the structured sort `Value`. `Value_Custom` value is used to represent Strings, `Custom_Value` is an enumeration sort containing a constant for every concrete String in the collection of state machines. The sort could in the future be easily extended with more complex data structures.

```
Value = struct
  Value_Int(Int)
  | Value_Bool(Bool)
  | Value_Custom(Custom_Value);
```

Example 3.1. *We consider a very basic action language consisting of the constants* `true` *and* `false` *and a variable assignment operator:* `V := e`*, where* `V` *is a* `VarName` *and* `e` *is either* `true` *or* `false`*. The following mCRL2 code defines all the necessary sorts and mappings.*

```
sort
  Value = Bool;
  Instruction = struct
    Assignment(VarName, Value)
    | Constant(Value);
  ExcFrame = struct ExcFrame(Instructions, List(VarValuePair), ValueStorage);
  ValueStorage = VarName -> Value;
var
  inst: Instructions;
  i: Instruction;
  upd: List(VarValuePair);
  vn: VarName;
  v: Value;
  vs: ValueStorage;
  b: Bool;
eqn
  initializeExcFrame(inst, vs) = ExcFrame(inst, [], vs);
  executeExcFrameCode(ExcFrame(Assignment(vn, v) |> inst, upd))
    = ExecuteExcFrameCode(
      ExcFrame(inst, upd <| VarValuePair(vn, v), setValue(vs,vn,v));
  executeExcFrameCode(ExcFrame([], upd, vs)) = ExcFrame([], upd, vs);
  checkPredicate([Constant(b)], vs) = b;
  isFinished(ExcFrame(inst, upd, vs)) = #inst == 0;
  globalValuation(ExcFrame(inst, upd, vs)) = vs;
  varUpdates(ExcFrame(inst, upd, vs)) = upd;
  resetVariableUpdates(ExcFrame(inst, upd, vs)) = ExcFrame(inst, [], vs);
  getValue(vs, vn) = vs(vn);
  setValue(vs, vn, v) = vs[vn -> v];
```

Suppose we have a concrete action language program we want to execute containing two consecutive variable assignments.

```
[Assignment(V1, true), Assignment(V2, false)]
```

To execute the program we need to initialise an execution frame and supply a valuation. Let `eval` be some `ValueStorage` that maps both `V1` and `V2` to `false`. We obtain the following execution frame.

```
ExcFrame([Assignment(V1, true), Assignment(V2, false)], [], eval)
```

Applying `executeExcFrameCode` to it will yield a transformed execution frame.

```
ExcFrame([], [VarValuePair(V1, true), VarValuePair(V2, false)],
  eval[V1 -> true])
```

Using the mapping `isFinished` we can see the execution frame has finished executing the code. We can extract the updated valuation with the mapping `globalValuation` and the assignments to variables with `varUpdates`. Note that if we wanted to execute only one instruction at a time (because we want to allow more interleavings) we could simply update `executeExcFrameCode`.

3.4 Representing State Machines in mCRL2

In this section we present how the structural elements of a state machine are represented in our generic mCRL2 model. In particular we describe various notions of states and transitions.

We assume that `StateName` and `CompName` have been declared as mCRL2 enumeration sorts, enumerating, respectively, all state names and all state machine identifiers occurring in the SysML model under consideration.

We proceed by introducing the sort `StateInfo`, which is either a triple with constructor `SimpleState` or with constructor `CompositeState`, both with projection functions `parent`, `entryAction` and `exitAction`, or it stores a single data element together with a constructor (`JoinVertex`, `JunctionVertex`, etc.).

```
StateInfo = struct
  SimpleState(
    parent: StateName,
    entryAction: Instructions,
    exitAction: Instructions)
  | CompositeState(
    parent: StateName,
    entryAction: Instructions,
    exitAction: Instructions)
  | JoinVertex(parent: StateName)
  | JunctionVertex(parent: StateName)
  | ForkVertex(parent: StateName)
  | InitialState(parent: StateName)
  | FinalState(parent: StateName)
  | ChoiceVertex(parent: StateName);
```

The parent of a state is stored to represent the hierarchy of states induced by composite states. A state's parent is the first enclosing composite state. We assume that the sort `StateName` has a special element `root`; states that are not enclosed in a composite state have `root` as their parent. For example the states 'Initial1' and 'Starting Controller' in Figure 3.3 respectively have the following `StateInfo`.

```
InitialState(root)
SimpleState(Booting, [], [])
```

Our framework supports change events and timeout events, see the definition of the sort `Event` below. The event type `none` is reserved for transitions without a trigger. Time is currently not modelled explicitly in our framework, even though mCRL2 does support it. Explicit timing would result in a significantly larger state space, while it is often not relevant for the verification of properties; indeed, this is the case in the context of EULYNX. Instead, transitions with a timeout event as trigger can fire non-deterministically. The generation of change events is discussed at the end of Section 3.6.

```
Event = struct none
  | ChangeEvent(getTriggerExpr:Instructions)
  | TimeoutEvent;
```

The sort `Transition` (given below) is used to specify the transitions of a state machine. The Boolean `internal` is used to differentiate between selfloops and internal transitions. The latter do not induce entry and/or exit behaviour.

```
Transition = struct Transition(
  source:StateName,
  trigger:Event,
  guard:Instructions,
  effect:Instructions,
  target:StateName,
  internal:Bool);
```

We also define the sort `StateMachine`, which aggregates all the information we need of a state machine.

```
StateMachine = struct StateMachine(
  transitions:List(Transition),
  initialState:StateName,
  states:List(StateName),
  stateInfo: StateName -> StateInfo,
  initialValuation: VarName -> Value);
```

The `initialState` designates the initial state in the root of the state machine (i.e. the initial state that is not contained in a composite state). The projection functions `states` and `stateInfo` retrieve which states are present in the state machine and the associated `StateInfo`, respectively. The function `stateInfo` only needs to be defined for the state names that occur in that state machine.

Due to the hierarchy of states, a state machine is ‘in’ a collection of states, a state configuration. A state configuration can be represented as a tree structure where the top node is not enclosed in a composite state. Parallel regions introduce nodes with multiple children. The mCRL2 excerpt below gives the definition of state configurations in the model.

```
StateConfig = struct StateConfig(
  rootState:StateName,
  substates:List(StateConfig));
```

An example configuration of the state machine depicted in Figure 3.3 is

```
StateConfig(
  Booting, [
    StateConfig(Initial2,[]),
    StateConfig(Initial3,[])
  ]).
```

3.5 Preprocessing Transitions

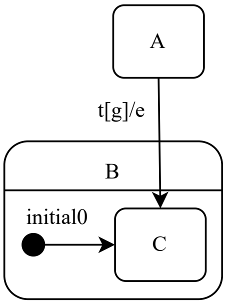
As explained in Section 3.1, state machines make a step from one state configuration to another. Such a step may consist of firing multiple transitions, which is the case with junctions, joins and forks. We could in theory perform step selection by performing a reachability analysis on the transitions in mCRL2, starting from the transitions that stem from the states of the current state configuration. We anticipate that this would make step selection computationally expensive. Instead, we opt to preprocess `Transitions` into `Steps`, eliminating junctions, forks and joins. The definition of `Step` is given below, as well as the mapping that derives `Steps` from `Transitions`. The `effect` of the step is a `ComposedBehaviour`. It allows us to create a partially ordered set of behaviour, which is needed for defining steps in the context of parallel regions (see Figure 3.5). A helper mapping transforms a list of `Instructions` to a `ComposedBehaviour`.

```
sort
  Step = struct Step(
    source: StateConfig,
    trigger: Event,
    guard: List(Instructions),
    effect: ComposedBehaviour,
    target: StateConfig,
    internal: Bool,
    arrowEnd: StateName);
  ComposedBehaviour = List(InstructionOrPar);
  InstructionOrPar = struct
    Instruction(getInstruction:Instruction)
    | ParBehaviours(parBehaviours:List(ComposedBehaviour));
map
  transitionsToSteps: StateMachine -> List(Step);
  InstructionsToComposedBehavior: Instructions -> ComposedBehavior;
```

The mCRL2 code specifying the transformation from `Transitions` to `Steps` consists of over 200 lines. Avoiding too much detail we illustrate which transformations are done.

From States to State Configurations

The first transformation using mCRL2 equations is to create a `Step` object for every `Transition` by adding the ancestors to the source and target state (see Figure 3.4). For simple transitions, of which the transition's start and end are not pseudostates, no more transformations are necessary. Note that we leave out parallel regions in defining transitions when they do not actively contribute to the step. If we were to include all the parallel regions in the `source` and `target` of `Steps` we would have to compute all combinations.



```
Transition(
  A,
  t,
  g,
  e,
  C,
  false)
```

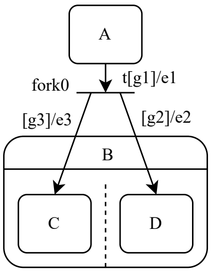
```
Step(
  StateConfig(A,[]),
  t,
  [g],
  InstructionsToComposedBehavior(e),
  StateConfig(B,[C]),
  false,
  C)
```

Figure 3.4: Example step between simple states.

Forks and Joins

A fork has one incoming transition and multiple outgoing transitions to states in different parallel regions of a composite state. The incoming and outgoing transitions should be traversed atomically and can only be fired when the guards on all the transitions are true. The action language behaviour of the outgoing transitions is performed in parallel. We preprocess forks into steps by combining the incoming and outgoing transitions (see Figure 3.5). Note that in the case of a fork the step does not have a single `arrowEnd`. We assume that `StateName` has a special element `multiple` which is used in the case of forks.

A join has multiple incoming transitions from states in different parallel regions and has one outgoing transition. The semantics and preprocessing of joins is symmetrical, the incoming and outgoing transitions and their guards are combined. This transformation is only correct under the assumption that both the incoming and outgoing transitions do not, respectively, come from or go to other pseudo states. In other words, we do not allow the combination of several joins or forks, or combinations of joins/forks and junctions.

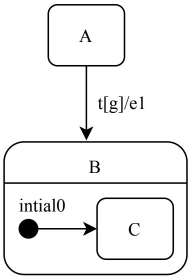


```
Step(
  StateConfig(A,[]),
  t,
  [g1,g2,g3],
  e1 ++  [ParBehaviours(e2,e3)],
  StateConfig(B,
      [StateConfig(C,[]),
      StateConfig(D,[])])
  false,
  multiple)
```

Figure 3.5: Example steps to and from a fork.

Adding Initial States

Steps ending in a composite state should start in the corresponding initial state. In a preprocessing step we add the initial state in the target (see Figure 3.6).

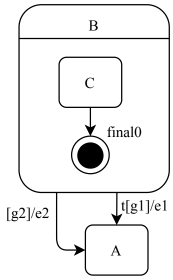


```
Step(
  StateConfig(A,[]),
  t,
  [g],
  e1,
  StateConfig(B,[initial0]),
  false,
  B)
```

Figure 3.6: Example step to a composite state.

Adding Final States

Final states indicate that the enclosing state is finished. Transitions with a composite state as source without a trigger are only enabled when every parallel region of the composite state is in a final state. For transitions with a trigger there is no such requirement. We preprocess steps from composite states without a trigger by adding final states to the source state configuration (see Figure 3.7).



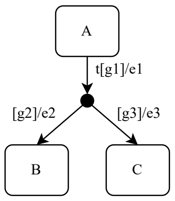
```
Step(
  StateConfig(B,[final0]),
  none,
  [g2],
  e2,
  StateConfig(A,[]),
  false,
  A)
```

```
Step(
  StateConfig(B,[]),
  t,
  [g1],
  e1,
  StateConfig(A,[]),
  false,
  A)
```

Figure 3.7: Example steps from a composite state.

Removing Junctions

Junctions may have multiple incoming and outgoing transitions. The interpretation is that a step can be taken from any of the states with a transition to the junction to any state with an incoming transition from the junction. The combined step is fired atomically, the guards of both the incoming and outgoing transition need to be evaluated beforehand. Junctions are removed in a preprocessing step by introducing a step for each path over the junction, combining the guards (see Figure 3.8). Junctions may be combined, i.e. transitions between junctions are allowed as long as there are no cycles.



```
Step(                     Step(
  StateConfig(A,[]),        StateConfig(A,[]),
  none,                     t,
  [g1,g2],                  [g1,g3],
  e1 ++ e2,                 e1 ++ e3,
  StateConfig(B,[]),        StateConfig(C,[]),
  false,                    false,
  B)                        C)
```

Figure 3.8: Example steps involving a junction.

3.6 Step Selection

Given a state configuration and a set of steps we can reason about which steps are enabled for firing. We go over the restrictions for firing steps, which are checked in several data equations.

The most basic requirement for selecting a step is that the source of the step must match the current state configuration. This is checked by `filterPossible`, defined below. The helper function `getAllStatesConfig` returns the set of all states that are in a state configuration. Another helper function called `containsPseudoState` checks whether a state configuration contains a pseudostate. Due to the run-to-completion semantics we only select a new step when we have reached a stable state configuration (i.e., a state configuration without pseudostates). For this reason we add the condition that if the current state configuration contains a pseudostate then we will only consider transitions from a pseudostate. The helper function `matchState` checks whether the source state configuration is contained in the current state configuration.

```
map
  filterPossible: List(Step)#StateConfig#StateMachine -> List(Step);
  matchState:StateConfig#StateConfig -> Bool;
var
  sc, sc1, sc2: StateConfig;
  step: Step;
  steps: List(Step);
  sm:StateMachine;
eqn
  filterPossible([], sc, sm) = [];
  filterPossible(step |> steps, sc, sm) = filterPossible(steps,sc,sm)
    ++ if(matchState(sc,source(step))
        && (containsPseudoState(sc,sm)
          => containsPseudoState(source(step),sm)),
     [step], []);
  matchState(sc1,sc2) = (getAllStatesConfig(sc2)
    - getAllStatesConfig(sc1))=={};
```

Another requirement is that the guard evaluates to true and the trigger matches the current event that is being processed. These two checks are performed by `filterEnabled`.

```
map
  filterEnabled: List(Step)#List(Event)#ValueStorage -> List(Step);
vars
  tr: Step;
  trs: List(Step);
  e_list: List(Event);
  vars: ValueStorage;
eqn
  filterEnabled([],e_list,vars) = [];
  filterEnabled(tr |> trs,e_list,vars) =
    if(trigger(tr) in e_list && forall g:Instructions.
      g in guard(tr) => checkPredicate(g,vars),[tr],[])
    ++ filterEnabled(trs,e_list,vars);
```

Another rule is that steps for which the source is lower (i.e. more deeply nested) in the state hierarchy have a higher priority than steps for which the source is higher in the state hierarchy. The mapping `filterPriority` selects the steps with the highest priority among the input. Note that there may be multiple steps on the same priority level, so the mapping may return multiple steps. The state machine makes a non-deterministic choice between these steps, as we will see in Section 3.7.

```
filterPriority: List(Step) -> List(Step);
```

As mentioned earlier, a state machine can also perform a multi-step if multiple steps with the same trigger event are enabled in parallel regions. To be more precise: the state machine selects a multi-step consisting of the maximal set of non-conflicting steps, where non-conflicting means that no two steps in the set exit the same state. The mapping `multiStepPossibilities` computes all such multi-steps given a set of steps.

```
multiStepPossibilities: List(Step) -> List(List(Step));
```

Due to the way we have constructed `Steps` the target field of a transition is not always a complete state configuration (unaffected parallel regions are missing). To construct the new state configuration `computeNextState` takes the target of a transition and adds the parallel regions of the current state configuration that are unaffected (i.e. not exited).

```
computeNextState: StateConfig#Step -> StateConfig;
```

Function `computeNextState` recurses through the tree structure of the current and target state configuration. For each encountered composite state it copies over unaffected parallel regions. A region is unaffected when the region was not present in the source of the step (it was not an active participant of the step) and it is not exited by the step.

The behaviour of performing a step, i.e. an instance of `ComposedBehaviour`, is the behaviour of the step itself combined with possible exit and entry behaviour. For internal transitions no state is entered or exited. The snippet below shows the definition of `determineBehaviourStep`.

```
map
  getEntryBehaviour: StateMachine#StateConfig#Step -> ComposedBehaviour;
  getExitBehaviour: StateMachine#StateConfig#Step -> ComposedBehaviour;
  determineBehaviourStep: StateMachine#Step#StateConfig -> ComposedBehaviour;
var
  sm: StateMachine;
  cur: StateConfig;
  st: Step;
eqn
  (!internal(st)) -> determineBehaviourStep(sm,st,cur) =
    getExitBehaviour(sm,cur,st)
      ++ effect(st)
      ++ getEntryBehaviour(sm,cur,st);
  internal(st) -> determineBehaviourStep(sm,st,cur) = effect(st);
```

Both `getEntryBehaviour` and `getExitBehaviour` compute the new state configuration after firing the transition and which states are entered/exited; subsequently they determine the order in which behaviour needs to be executed and construct a `ComposedBehaviour`. The order of composing entry behaviour is outside-in (top level states first) and the order of composing exit behaviour is inside-out (nested states first). To determine the order both functions recurse through the new state configuration. For states that are on the same level (in parallel regions) the action language behaviours are put in parallel.

We use a mapping `computeExecutionOptions` to compute all the options for what behaviour from a `ComposedBehaviour` can be executed next. If the head of the composed behaviour is a sequential composition of instructions it will return one execution option with all instructions up to the end of the composed behaviour or up to a parallel composition (whatever comes first). If the head of the composed behaviour is a parallel composition then we get multiple options corresponding to each parallel branch.

```
sort
  ExecutionOption = struct ExecutionOption(
    getCodeToExecute:Instructions,
    getRemainingBehavior:ComposedBehavior);
map
  computeExecutionOptions: ComposedBehavior -> List(ExecutionOption);
```

A change event is generated when the content of a `when`(x) trigger *becomes* true. When a variable is updated we need to check which change events need to be generated. For this purpose we introduce the sort `Monitor`. A monitor stores an action language expression and the previous evaluation result. When we update a variable we can check which change events are generated using `deriveChangeEvents`. The mapping `updateMonitors` updates the valuation stored in the monitors.

```
sort
  Monitor = struct Monitor(getExpression:Instructions, getValuation:Bool);
map
  deriveChangeEvents: List(Monitor)#ValueStorage -> List(Event);
  updateMonitors: List(Monitor)#ValueStorage -> List(Monitor);
var
  vars: ValueStorage;
  mon: Monitor;
  mons: List(Monitor);
eqn
  deriveChangeEvents(mon |> mons,vars) =
  if(checkPredicate(getExpression(mon),vars) && !getValuation(mon),
    [ChangeEvent(getExpression(mon))], []) ++ deriveChangeEvents(mons,vars);
  deriveChangeEvents([],vars) = [];
```

3.7 StateMachine Process

The state machine process uses the data operations that we described in earlier sections and uses them to specify the observable *actions* of a single state machine, which are visible in the LTS associated to the mCRL2 model. For now we will present a slightly simplified version, which we will extend when we incorporate SysML specific communication in Section 3.8. Below we present the parameters of the process and the declaration of the observable actions (which includes the parameters of those actions).

```
act
  discardEvent: CompName#Event;
  selectMultiStep: CompName#Event#List(Step);
  executeStep: CompName#Step;
  executeBehaviour: CompName;
proc
  StateMachine(
    ID:CompName,
    SM:StateMachine,
    sc:StateConfig,
    eq:List(Event),
    ms:List(Step),
    behav:ComposedBehaviour,
    mon:List(Monitor),
    vars:ValueStorage,
    exc:ExcFrame) = ...
```

The UML standard does not define in what order events are processed. Our modular approach allowed us to experiment with various options. In the end we opted to process events in FIFO order, hence the event queue `eq` is a list of events. When events were processed in arbitrary order we saw several safety concerns in EULYNX models. In particular, we saw that messages from a previous connection attempt could be processed at a later point, allowing a connection to be established while there was an error (such as a mismatching checksum).

The `StateMachine` process consists of an alternative composition where each summand performs one action and then recurses (with updated parameters). The observable actions are chosen to reflect decisions in the run-to-completion cycle.

When both `ms` and `behav` are empty a new multi-step should be considered. If no step is enabled by the head of the event queue the process can perform a `discardEvent` action and remove it from the event queue, as specified below. The mapping `filterPossible` checks which transitions are possible from the current state configuration. The mapping `filterEnabled` filters those transitions to transitions that have the specified event as trigger and whose guard evaluates to true. Hence, in the case that a transition has both a triggering event and a guard specified then both conditions need to be met before the transition can be fired. Note that mCRL2 allows for an abbreviated, assignment-like syntax in which only the parameters to be updated need to be mentioned in a recursive call; all other parameters of the process remain the same.

```
+ (#ms == 0 && #behav == 0 && exc == EmptyExcFrame
  && #eq != 0 && !containsPseudoState(sc,SM(comp))
  && #filterEnabled(filterPossible(steps,sc,SM(comp)), head(eq), vars) == 0)
    -> discardEvent(ID,head(eq)).StateMachine(eq = tail(eq))
```

Alternatively, we can select a multi-step with a `selectMultiStep` action.

```
+ (#ms == 0 && #behav == 0 && exc == EmptyExcFrame) ->
  (sum multi:List(Step). %step may consist of multiple transitions
    (
      multi in multistepPossibilities(
        filterPriority(
          filterEnabled(
            filterPossible(steps, sc, SM(comp)),
              [none,TimeoutEvent] ++ if(#eq > 0, [head(eq)], []), vars)))
    ) -> (selectMultiStep(id,trigger(head(multi)),multi)
      .StateMachine(
        eq = if(#eq > 0 && trigger(head(multi)) == head(eq),tail(eq),eq),
        ms = multi)))
```

We can now perform an `executeStep` action to start executing one of the selected steps, which updates `sc` and puts the composed behaviour of the step in `behav`.

```
+ (#behav == 0 && exc == EmptyExcFrame) -> sum n:Nat. (n < #ms)
  -> executeStep(id,ms.n).StateMachine(
    ms = removeSteps(ms,[ms.n]),
    behav = determineBehaviourStep(SM(comp),ms.n,sc),
    sc = computeNextState(sc,ms.n)
  )
```

The process selects one of the execution options calculated by the mapping `computeExecutionOptions` and initialises an `ExcFrame` which is stored in `exc`. The process calls `executeExcFrameCode` and performs an `executeBehaviour` action until the execution frame is finished. Every time code is executed (and thus possibly variables are updated), it is checked whether change events can be derived. When the execution frame is finished we compute a new execution option.

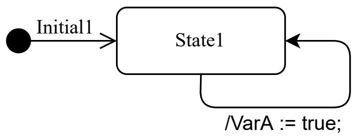
```
Statemachine(...) =
...
+ (#behav != 0 && exc == EmptyExcFrame)
  -> sum eo:ExecutionOption.
    (eo in computeExecutionOptions(behav) && #getCodeToExecute(eo) > 0) ->
      StateMachineExecuteCode(
        exc = initializeExcFrame(getCodeToExecute(eo),vars),
        behav = getRemainingBehavior(eo))
+ (exc != EmptyExcFrame) -> StateMachineExecuteCode();

%Auxilary process with the same parameters as StateMachine
StateMachineExecuteCode(...) =
  executeBehaviour(id).StateMachine(
  vars = globalValuation(executeExcFrameCode(exc)),
  exc = if(!isFinished(executeExcFrameCode(exc)),
    executeExcFrameCode(exc),
    EmptyExcFrame),
  eq = eq ++ deriveChangeEventsMultUpdates(mon,vars,
    varUpdates(executeExcFrameCode(exc))),
  mon = updateMonitors(mon,globalValuation(executeExcFrameCode(exc))));
```

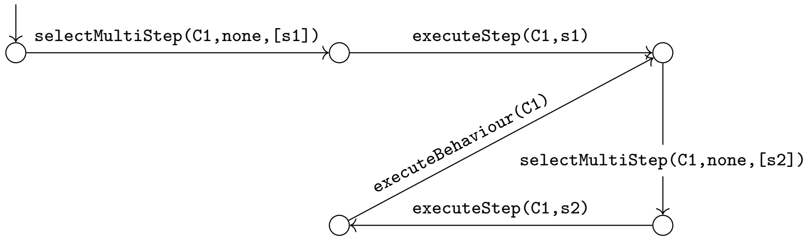
When the execution of `behav` is finished we select a next step from `ms`. When there are no more steps to execute the process is ready to select a new multi-step.

Depending on the kind of analysis that will be performed on the resulting LTS we might want different observable actions. If we would want to verify something regarding the state configuration we might want to add a self loop signalling the current state configuration. Alternatively, we might want to hide some actions by renaming them to τ, indicating that they are unobservable.

Example 3.2. *Consider the state machine below.*



The LTS induced by this state machine is given below, where `C1` *is the name of the component,* `s1` *is the* `Step` *from the initial state and* `s2` *is the* `Step` *representing the selfloop. For both steps the corresponding* `selectMultiStep` *action has a parameter* `none` *indicating that they do not require an event to trigger them.*



3.8 SysML Specific Communication

Specific to the EULYNX dialect of SysML is that there are ports over which communication takes place. The interfaces are specified by IDBs, which specify the ports of components and their connections.

As mentioned earlier, we focus on the semantics of a set of communicating state machines. We assume that we have the following communication structure between state machines. Each component has a set of ports, which are subdivided into input and output ports. An output port can be connected to multiple input ports. Both input and output ports need not be connected at all, in which case they are intended to interact with the environment. One more assumption on the action language is that ports are treated as variables: changing the variable associated to an output port leads to a communication, which updates the variable associated to the input port of the receiver.

The sort `Component` extends state machines with extra information. The sort `CompName` defines a finite enumeration of identifiers for components. The sort `CompPortPair` forms a unique identifier for a port, combining the name of a port and the name of the component it belongs to.

```
CompPortPair = struct CompPortPair(getComp: CompName, getPort: VarName);
Component = struct Component(
  name: CompName,
  SM: StateMachine,
  in_ports: List(VarName),
  out_ports:List(VarName));
```

To take communication between state machines into account, we modify the `StateMachine` process of Section 3.7 by replacing the state machine parameter with a comp parameter, adding a message queue parameter mq and adding some extra actions:

```
act
  sendComp,sendI,send: CompPortPair#Value;
  receiveI,receiveComp,receive: CompPortPair#Value;
proc
  StateMachine(...,comp:Component, mq:List(VarValuePair)) = ...
```

When executing an execution frame it is checked whether there are updates to output ports, which are then stored in output queue mq. The call to StateMachine in StateMachineExcetuteCode has the following addition.

```
mq = distillPortUpdates(varUpdates(executeExcFrameCode(exc)),comp,vars)
```

The mapping distillPortUpdates uses the port information in comp to check which variable updates are of ports (instead of local variables).

When mq is not empty the process can perform a sendComp action, communicating the update.

```
+ (#mq > 0)
  -> sendComp(CompPortPair(id, getVariable(head(mq))),getValue(head(mq)))
    .StateMachine(mq = tail(mq))
```

At any point in time the process can receive messages via a receiveComp action.

```
+ sum v:Value,p:VarName. receiveComp(CompPortPair(ID,p),v)
  .StateMachine(vars = setValue(vars,p,v),
    eq = eq ++ deriveChangeEvents(mon, setValue(vars,p,v)),
    mon = updateMonitors(mon, setValue(vars,p,v)))
```

We want to ensure that when a value is sent on an output port, it is received by all (and only) connected input ports. This is enforced by the Messaging process and the allow and communication operators in the initialisation process. The Messaging process is specific to a particular configuration and contains a summand for each communication channel. Below we give the initial process expression and an example of an instantiation of the Messaging process in which port P1 is connected to P2, and P3 is connected to both P4 and P5.

```
proc Messaging =
  sum v: Value. sendI(CompPortPair(C1,P1),v)
    |receiveI(CompPortPair(C2,P2),v).Messaging
  + sum v: Value. sendI(CompPortPair(C1,P3),v)
    |receiveI(CompPortPair(C2,P4),v)
    |receiveI(CompPortPair(C3,P5),v).Messaging
init allow({selectMultiStep, discardEvent, executeStep, executeBehaviour,
    send|receive, send|receive|receive},
    comm({
      sendComp|sendI -> send,
      receiveI|receiveComp -> receive},
        Messaging||Environment
        ||StateMachine(...)||StateMachine(...) ...));
```

The central idea is that individual components need not know how ports are connected. Instead, Messaging provides a 'meeting place' with which the sender and receivers synchronize. As an example, suppose some component $C1$ sends some value v on port $P1$ that should be received by two receivers $C2$ and $C3$ on ports $P2$ and $P3$, respectively. The Messaging process and the StateMachine process of the sender and the two receivers can perform the *multi-action*

```
sendComp(C1,P1,v)|receiveI(C1,P1,v)|sendI(C2,P2,v)
|sendI(C3,P3,v)|receiveComp(C2,P2,v)|receiveComp(C3,P3,v).
```

This is transformed by the communication operator to `send(C1,P1,v) |receive(C2,P2,v)|receive(C3,P3,v)`.

Ports that are not connected to any other port are exposed to the environment, i.e., adjacent systems not included in the model. Input ports exposed to the environment can expect inputs at any moment in time. We model the environment with the `Environment` process, which depends on the configuration, just like the `Messaging` process. An example of an instantiation is given below. For each port open to the environment (input ports as well as output ports) a summand is needed. For input ports a restriction is needed to enforce that only values of the correct data type can be sent. Note that a connection between the environment and an exposed port must also be created in the `Messaging` process.

```
Environment =
  %output port, environment receives anything
  sum v: Value. receiveComp(CompPortPair(ENV_C1, P1), v).Environment()
  %boolean input port
  + sum v: Value. (v in [Value_Bool(true), Value_Bool(false)])
    -> sendComp(CompPortPair(ENV_C1, P2), v).Environment()
```

3.9 Configuration and Automatic Translation

In the previous sections, we have discussed the generic parts of the mCRL2 model; in this section, we describe how to configure the model with a specific configuration.

First, the enumerations `StateName`, `CompName` and `VarName` need to be instantiated. The action language needs to be defined: the sorts `Value` and `Instruction` need to be defined. Also the semantics of the action language needs to be defined by extending the sort `ExcFrame` and giving defining equations for the mappings listed in Section 3.3. Finally, the initial process expression needs to be given, in accordance with the structure described in Section 3.8. The `Environment` and `Messaging` processes must be instantiated appropriately. For every state machine an instantiation of the `StateMachine` process needs to be added.

The models of the case studies, from Chapter 5, are available in the Zenodo repository [12]. They are concrete instantiations of the generic model.

Manually configuring the generic mCRL2 model with a configuration would be time consuming and error prone. Instead, we have created a tool named `sysml2mcrl2`, which automatically creates an mCRL2 model based on a collection of SysML diagrams. This tool is implemented in Java.

Unfortunately, EULYNX SysML models, which are created in the tool PTC Windchill modeller, cannot be exported to a format that is easy to import in other tools. There is an XML export option but the output does not conform to the OMG metamodel. Instead, EULYNX SysML diagrams need to be transcribed to jEULYNX, an internal Java DSL specifically made for our purposes. This DSL has been designed and implemented by Djurre van der Wal, who is also part of the FormaSig project.

The tool `sysml2mcrl2` has three main features:

- It performs a number of sanity checks on the input to detect inconsistencies within and between diagrams;
- It combines all the IBDs and produces a global overview of the connections between ports.
- For ports that are not connected to any other port (they are open to the environment), it allows the user to specify the initial value and restrictions on what values can be received.

We will not explain the internal workings of jEULYNX or `sysml2mcrl2`. The source code is available in the Zenodo repository accompanying this thesis [12].

3.10 EULYNX Adaptations to the mCRL2 Formalisation

To accommodate verification of EULYNX models we have enriched the mCRL2 model that encodes the semantics of SysML state machines. We needed to adjust the semantics to accommodate pulse ports, which are Boolean-valued ports with a semantics deviating from other data types. Suppose that we have some pulse output port X connected to input port Y. When a state machine sets port X to the value true, then port X will *automatically* revert to false after a brief time, and so will Y. The exact semantics of this mechanism is imprecise in EULYNX specifications. Our models do not model time explicitly, but we can capture the semantics of pulse ports in an alternative way. Our interpretation is that X and Y are true only in one atomic moment, during the communication from X to Y. The sender and receiver may use pulse ports to trigger *change events*: if the receiver has a transition with trigger 'when(Y)', a change event is placed in the event queue.

Variables, including ports, should have an initial value as otherwise it might not be possible to evaluate ASAL expressions referencing those variables. In EULYNX, all output ports and local variables are initialised in the transition from the initial state in the root of the state machine. Input ports can then be initialised by looking up the initial value of the connected output port. However, ports need not be connected, because they can interact with the environment (systems outside the scope). Input ports open to the environment are therefore not initialised in EULYNX specifications. The `sysml2mcrl2` tool allows manual specification of the initial value of these ports.

For input ports open to the environment it is also not clear what values can be received while the system is running. Our default interpretation is that any value of the data type of the port is allowed. The `sysml2mcrl2` tool also allows us to disable or restrict selected input ports. For ports with data type 'String' a finite number of concrete strings needs to be provided.

3.11 Concluding Remarks

One of the main benefits of our generic formalisation of the semantics of SysML in mCRL2 is that it facilitates a straightforward automated translation. To have an automated translation from SysML to mCRL2 we only need to implement a tool that extracts the configuration data from a SysML model and outputs the mCRL2 code as described in Section 3.9. The implementation of this tool will not be discussed in this thesis.

The UML standard does not give guidelines about the degree of interleaving in the execution of action language expressions. This ambiguity affects both the interleaving between state machines and the interleaving between parallel action language behaviours in a step. A choice has to be made, presenting the following trade-off. The finest mode of interleaving would break action language behaviour execution down to single instructions (such as looking up the value of a variable). Choosing this interleaving model would allow the most detailed analysis. The coarsest mode would implement a run-to-completion semantics for parallel behaviour, reducing the state space.

A benefit of our approach is that we can easily experiment with variations on the semantics. The variation of the interleaving of action language behaviour can for example easily be achieved by modifying the `ExecuteExcFrameCode` mapping. We have also experimented with the event queue; concluding that a FIFO order is important for the safety of EULYNX (see Section 3.7).

A final benefit of directly formalising in mCRL2 (compared to formalising in plain mathematics) is that the mCRL2 toolkit acts as an IDE. The parser and type checker of the editor point out the most basic mistakes. Moreover, the model can be simulated when provided with a configuration of a simple set of state machines. This provides an additional way of testing whether the semantics is as intended.

Concluding, we have shown how we have formalised the semantics of (SysML) state machines directly in mCRL2. The generic mCRL2 model is flexible and could be adjusted for a wide range of action languages. The step to an automated translation using our model is small and has been achieved in FormaSig.

Chapter 4

Scalability

Great is our LORD, and of great power: his understanding is infinite.

Psalm 147:5, KJV

The mCRL2 toolset [24] has been used in various industrial applications [21, 73, 108]. A recurring challenge is that industrial models are often complex and consist of many concurrent components. Hence, the state spaces induced by such models are typically huge, hindering verification efforts. The size of the state space tends to scale exponentially with the number of components. This is a well known phenomenon often referred to as the state space explosion problem.

For EULYNX models we face the same challenge. EULYNX SysML models typically consist of 6-10 parallel components. When we apply the translation defined in Chapter 3 we obtain models that are not amenable to formal verification. The least complex EULYNX SysML models induce a state space consisting of several billions of states whilst for more complex models we cannot even compute the size of the state space. Techniques are needed to reduce the state space before we can even consider model checking.

One of the main causes of the state space explosion with mCRL2 models obtained with our SysML-to-mCRL2 translation is the communication between components (and messages arriving at ports open to the environment). Most communication is asynchronous; the sender sends a value along a port, which triggers a change event at the recipient, this event is then stored in an event queue until it is either used to trigger a transition or discarded. Models can have input ports which are not connected to an output port and thus open to the environment. Every state in the state space has transitions representing a message arriving from the environment. Without limiting the environment in sending messages the state space induced by FormaSig models is infinite as the environment can always add one more event to the event queue of a component. The mCRL2 toolset can only perform model checking for finite state spaces, though there is some research into model checking with infinite data domains in mCRL2 [91]. The state space can be restricted by bounding the event queue, disallowing reception of messages until some events are processed. The downside of this approach is that some behaviour that was permitted by the original SysML model is not present in the mCRL2 model and can therefore not be analysed during model-checking.

As SysML is not formal and allows for multiple interpretations there is some degree of freedom in how we define the semantics. Some interpretations lead to bigger state spaces than others. We can reduce the state space by adapting the formalisation as presented in the previous chapter. Additionally, we may be able to make some assumptions on what behaviour is possible from the environment. We use the following techniques to reduce the state space, which are discussed in more detail later in this chapter:

1. Disable input ports open to the environment that are actually configuration parameters. Some ports represent configuration parameters, such as the length of a certain timeout, we set these ports to a fixed value.

2. Abstract from specific values for input ports open to the environment. For some input ports multiple values can be received but they are all treated in the same way (mostly, they are sent as-is over the EULYNX connection). For these ports we abstract from specific values and use a dummy value.

3. Bundle messages on pulse ports and their accompanying data parameter ports.

4. Reset data parameter ports to a dummy value when the pulse has been handled.

5. Restrict the environment by disabling inputs when the total number of events in the system is above a certain threshold.

6. Split the SysML models into separate groups of components and analyse them separately.

The first four techniques do not meaningfully change what behaviour is possible. The last two techniques, however, do. By restricting the environment we lose many states that are, according to the specification, reachable. By splitting models we lose how the two parts restrict each other's behaviour. This is necessary to still get some partial information about the quality of the models.

Another way to reduce the size of the state space, is to minimise it modulo an equivalence relation, such as strong bisimulation [100] or branching bisimulation [52]. For some models the state space induced by the model is too large to generate, while the size of the minimised state space is small enough to allow model checking. For such models *compositional minimisation* is a helpful technique. As the name suggests, the technique applies minimisation not at the top level but at the level of components. The model is split into several submodels representing the components of the system. The state spaces of the components are minimised and then combined to construct a behaviourally equivalent but smaller state space.

Compositional minimisation is not a novel technique and has a history going back to the early 1990s [53]. The CADP toolset [42] includes the tool EXP.OPEN, which computes the parallel composition of a network of state spaces. These existing techniques are not directly applicable in the context of mCRL2, since they do not support the mCRL2 concept of multi-action.

In [79] a technique is described to maximally minimise during compositional minimisation. They analyse the actions that occur in the parallel components of a process algebraic specification and in a given μ-calculus formula. Based on this information it is determined which actions can be hidden, which components can be minimised modulo divergence preserving branching bisimulation and which components must be minimised modulo strong bisimulation. This technique is implemented on top of the CADP toolset.

Many papers on the topic of compositional minimisation focus on interface specifications, which model the interfaces between the separated components [43, 54, 78]. They solve a common complication with compositional minimisation: the state spaces of the components can be much larger than the state space of the entire system. We observed this phenomenon for our models as well, but it was solved by techniques 3 and 4 (mentioned above), which bundle pulses and their data parameters in a single message (see Section 4.1.2 for more information).

Compositional state space generation has recently been added to the mCRL2 toolset in the form of the tools `lpscleave` and `lpscombine` that, respectively, decompose and recombine a model [83]. Cleaving an LPS P produces two new LPSs, L and R, both containing part of the behaviour. The LPS is cleaved based on a partitioning of the data parameters of P, specified by the user. Subsequently the state spaces associated to L and R can be computed and minimised with respect to strong bisimilarity, yielding, say, L' and R'. The tool `lpscombine` combines L' and R' to the final state space, ensuring that the result is strongly bisimilar to P.

One of the complications these tools have to overcome is dealing with the multi-action semantics of mCRL2 [55]. Since τ is the identity element for multi-actions, i.e. $a|\tau = a$, τ actions can arbitrarily communicate with other components in a parallel composition. If components had τ-transitions, `lpscombine` would be unsound. To prevent this phenomenon, `lpscleave` replaces all occurrences of τ's by a visible action `tag` in L and R. The recombination process of `lpscombine` abstracts again from this visible action.

Many behavioural equivalences (such as branching bisimilarity) include a special treatment of τ-transitions [45]. If a process has many τ-transitions, like our FormaSig models, then minimisation with respect to notions of bisimilarity that treat τ-transitions as unobservable yields a much smaller state space. Since, by construction, L and R do not contain τ-transitions we cannot effectively use minimisation with respect to branching bisimilarity on the level of components.

Our contribution is an extension of the theory of [83] in which we add support for intermediate branching bisimulation minimisation. In short, this is achieved by first hiding the `tag` action labels in the cleaved processes (turning many transitions into τ-labelled transitions), then minimising the state space and finally reintroducing the `tag` labels that were hidden in the first step (to avoid communicating τ-actions).

In effect, we treat tag as unobservable whilst computing the branching bisimulation quotient. The extension is proven correct. The techniques can be generalised to other process algebras using multi-actions.

Compositional minimisation turns out to be a very effective technique to minimise FormaSig models. In our models there is a natural notion of component (a state machine), and, due to the way the semantics of state machines is defined in mCRL2, these components exhibit quite some internal behaviour. Hence, a significant reduction of the size of the state spaces associated with the components can be achieved by branching bisimulation minimisation. The state space of recombining is several order of magnitude smaller than the original state space associated to the model.

The rest of this chapter is organised as follows. Section 4.1 presents the various techniques to reduce the state space by adapting the semantics. Section 4.2 summarises the cleave and combine method. In Section 4.3 the theory is extended to support branching bisimulation. Section 4.5 presents a number of benchmarks to measure the effectiveness of our state space reduction techniques. The chapter is concluded by Section 4.6, which summarises the achievements and presents some future work.

4.1 Adapting the Translation to Reduce the State Space

As mentioned in the introduction we found a number of techniques to reduce the state space by adapting the model. The techniques are diverse; some adapt the generic interpretation of SysML whereas others allow the modeller to add assumptions on the environment that are specific to the SysML model.

We start with the latter type in Section 4.1.1, where we explore how we can restrict what messages can arrive at input ports open to the environment. In Section 4.1.2 we present an adaption of the generic semantics in which a pulse and its associated data parameters ports are combined in a single message. Section 4.1.3 discusses limits on the event queues and assumptions regarding the synchronicity of communication. Finally, in Section 4.1.4 we present how we can abstract from certain components and how that affects what behaviour is possible.

4.1.1 Disabling Configuration Ports and Abstraction

Ports that are not connected to any block within the modelling scope of the system are open to the environment. For example, since the core interlocking logic is not part of EULYNX, ports modelling the interface to the core interlocking are open to the environment.

Without specific domain knowledge we cannot safely make assumptions on the behaviour of the environment. By default we can expect any value on input ports open to the environment in accordance with the type of the port. For example, a port with type ‘String’ could expect any string. However, in some cases we can (and should) restrict what kind of values are to be expected from the environment. For example, in the EULYNX point interface the commanded position can only be ‘left’ or ‘right’. Some input ports may actually represent parameters of the system that cannot change during execution, e.g. the EULYNX level crossing interface has a parameter indicating whether it should close after a connection timeout (the setting may differ between countries). We have added the ability to restrict inputs from the environment in our translation framework. The user can provide a list of concrete values that can be sent to a specific port.

In other cases the exact value that is received from the environment is irrelevant as they are all handled in the exact same manner, in which case we can abstract from the value and substitute it with a dummy value. As an example, the EULYNX SCI-LX level crossing interface can receive a number of distinct commands on the same port from the core interlocking which the modelled system simply forwards as-is to the field element (which is outside the scope of EULYNX). We do not want to disable the port as we do want to verify that the command is always delivered but we can abstract from the specific value.

4.1.2 Pulse Packs and Resetting Parameter Ports

In the SysML variant supported by our translation, messages with data fields are modelled using multiple ports. Data parameter ports (prefixed with ‘DT’) represent the data parameters. After all data parameter ports are set to the correct value by the sender, a pulse is sent on a pulse port (prefixed with ‘T’).

We observed that setting these data parameters separately significantly increases the state space, for two reasons. As each communication is a separate transition in the LTS, many interleavings occur with other behaviour in the system. Moreover, both the sender and the receiver store the value that was last sent on each data parameter port, even when this value is no longer relevant. A good example of the effect is in the block S_SCI_LC_SR of the EULYNX SCI-LC level crossing interface[1]. The block essentially forwards messages (with data parameters) between the core interlocking and the level crossing object controller. The following internal transition is from the state TRANSMIT_COMMANDS_OR_MESSAGES of the S_SCI_LC_SL block (to improve readability we shortened the port names):

```
when(T105_Status)/
DT5_Status := DT105_Status;
T5_Status := TRUE;
```

Two transitions in the LTS are needed to send what is essentially a single message. The values of DT5_Status and DT105_Status are remembered (until a new message is processed) even though the information is no longer relevant.

To optimise our model we introduced the concept of *pulse pack*, which is a composite message where all the data parameters are packed into a pulse on a pulse port. This change removes the interleavings that were present in the original model, which are unrealistic for packet based communication anyway. In the

[1]The relevant diagram can be found on page 26 of EULYNX document Eu.Doc.108 v1.0.

mCRL2 model the `Value` struct is extended with a constructor for pulse packs which contains a list of data parameter port names and their assigned values.

```
Value = struct
    ...
    | Value_Pulse_Pack(params: List(VarValuePair)) ?is_pulse_pack
```

An additional optimisation is that we reset the data parameters to their initial value when they are no longer relevant. The sender resets its data parameter ports at the moment the pulse pack is sent. The receiver stores the values of the data parameter ports with the change event. The values are discarded when the event is discarded, or the transition, that was triggered by the event has finished executing.

Whether combining the pulse and the associated data parameters and resetting the parameter ports meaningfully changes the behaviour depends on how these ports are used. If a component uses the value of a data parameter port even when there is no accompanying pulse, then the transformation is invalid. In the context of EULYNX it is valid, since the modelling standard specifies that all data parameters should be set before a pulse is sent, and the value of the data parameter ports is only valid at the moment the pulse is received. In the future, EULYNX intends to replace pulse ports and data parameter ports with signals, which also combine all values in a single message. Resetting data parameter ports to a default value is valid since the modelling standard forbids using the values of data parameter ports later on. It specifies that if the value of a data parameter port is to be remembered for later use, then it should be stored in a local variable of the state machine (prefixed with 'Mem'). If such rules are not present, a static analysis could be performed to determine for which ports it is safe to use pulse packs and reset the parameter ports.

4.1.3 Buffered Communication and the Environment

Without limiting the environment in sending messages, the state space induced by FormaSig models is infinite as the environment can always add an extra event to the event queue of a component.

A simple solution is to put a bound on the event queues; a component can then only receive a message if the event queue is not full. A downside of this approach is that the model no longer corresponds to the SysML specification. Additionally, it can introduce deadlocks. When two components both have a full event queue and are in a run-to-completion in which they need to send a message to each other they can no longer make progress.

A way to avoid these deadlocks is to (also) restrict the environment. By restricting when the environment can send a message it is possible to make the state space finite without introducing deadlocks. We have added an optional restriction on the environment that it can only send a message when the total number of messages in the system is below a certain threshold. This is implemented in the mCRL2 model using multi-actions. Sending a message from the environment is performed in synchronisation with all the state machines. The state machines send the length of their event queue and the environment has a guard restricting the sum of these lengths.

Communication over ports in SysML is asynchronous: the sending component can always send and the receiver processes places the result of the communication in the event queue. For communication over a network (between the object controller and the interlocking) asynchronous communication is the only logical option. However, within a physical component synchronous communication could also be achieved. Our framework allows marking some ports as synchronous. For synchronous ports the sending component blocks until the receiver is able to communicate. When the communication takes place the receiving component immediately selects a step based on change events or discards the change events when they do not enable any transition. This form of communication bypasses the event queue, decreasing the state space.

An option that is particularly interesting is to mark input ports open to the environment as synchronous. This way there is no sending party that needs to block further execution and we prevent putting events in the event queue that will be discarded in the next step anyway.

These optimisations change the semantics and influence the verification. The altered model contains a subset of the traces that are allowed by the SysML model.

4.1.4 Splitting Models

As the state space tends to grow exponentially with the number of components it is profitable to try to reduce the number of components in the analysis. SysML models are modular as the components behave independently, only interacting by message passing. We can abstract from certain components and remove them from the SysML model. The ports that are disconnected by removing a component become open to the environment.

For EULYNX models there is a natural way of splitting the components as each interface consists of a generic part and an interface specific part. They can be considered as different layers of the EULYNX protocol, which are only loosely coupled with few interactions between the layers. The requirements that we have identified for the various EULYNX interfaces can all be verified by only considering the components of one of the layers.

Since ports open to the environment are maximally permissive we do not lose any behaviour. We would have weak trace inclusion from the complete model to a partial model if we were to hide all the transitions belonging to behaviour that is not in the partial model. For every trace in the complete model there is a similar trace in the partial model, containing only the actions in which the components of the partial model participate.

We will now make it more precise when a component participates and characterise the trace inclusion. Let $\mathcal{C}$ be the set of components of the complete model. Let $\mathcal{P} \subseteq \mathcal{C}$ be the set of components in the partial model. Any transition has a set of *participants*. For any transition there are two cases:

- If a component performs an internal transition from the set below the component is the only participant.

$$\{\texttt{discardEvent}, \texttt{selectMultiStep}, \texttt{executeBehaviour}, \texttt{executeStep}, \\ \texttt{inState}, \texttt{inEventPool}, \texttt{resetVariables}, \texttt{varVal}\}$$

- If components communicate by sending a value along a port, the sending component and all the receivers participate.

For an action in the trace of the complete model there are three options:

- All participants are in $\mathcal{P}$: the same action is in the trace of the partial model.
- None of the participants are in $\mathcal{P}$ (all participants are components from which we have abstracted): the action is not in the trace of the partial model.
- Some of the participants are in the partial model and some of the participant are not. This is only the case when there is a communication between components that are in the partial model and components that are not. In this case a similar action is in the trace of the partial model; the `send`/`receive` contribution of the participant that is not in the partial model is replaced by a `send`/`receive` of the environment.

Trace inclusion holds only in one direction; the partial model may contain more traces than the original model. Components restrict each other's behaviour. By making some components part of the environment these restrictions are lifted as the environment is maximally permissive. Splitting models may then remove deadlocks present in the combined behaviour.

4.2 Compositional Minimisation

We proceed by exploring compositional minimisation in mCRL2, starting with an overview of the results of an earlier paper defining the theory of cleaving and combining [83]. Note that this section relies heavily on the definitions given in Section 2.2. This section defines a way to split an LPE (see Section 2.2.3), provides requirements on this split, defines a recombination process and provides a theorem stating that the recombined process is strongly bisimilar to the original process. In the next section we extend the theory to branching bisimulation.

Example 4.1. *Consider the following LPE, which we use as a running example. Note that states $P(0, \mathsf{false})$ and $P(1, \mathsf{false})$, and states $P(0, \mathsf{true})$ and $P(1, \mathsf{true})$ are branching bisimilar.*

$$\begin{array}{rl} & P(m : \texttt{Nat}, n : \texttt{Bool}) = \\ 0: & (m \approx 0) \to \tau \,.\, P(1, n) \\ 1: & + (m \approx 1) \to \mathsf{a} \,.\, P(2, n) \\ 2: & + (m \approx 1) \to \tau \,.\, P(2, n) \\ 3: & + (\neg n) \to \mathsf{b} \,.\, P(m, \mathit{true}) \\ 4: & + (m \approx 1 \wedge \neg n) \to \mathsf{c} \,.\, P(2, \mathit{true}) \end{array}$$

$$\begin{tikzcd}
P(0, \mathit{false}) \arrow[r, "\{\mathsf{b}\}"] \arrow[d, "\{\tau\}"'] & P(0, \mathit{true}) \arrow[d, "\{\tau\}"'] \\
P(1, \mathit{false}) \arrow[r, "\{\mathsf{b}\}"] \arrow[d, bend right, "\{\mathsf{a}\}"'] \arrow[d, bend left, "\{\tau\}"] \arrow[dr, "\{\mathsf{c}\}"] & P(1, \mathit{true}) \arrow[d, bend right, "\{\mathsf{a}\}"'] \arrow[d, bend left, "\{\tau\}"] \\
P(2, \mathit{false}) \arrow[r, "\{\mathsf{b}\}"] & P(2, \mathit{true})
\end{tikzcd}$$

Suppose we want to split it into two LPEs, of which one controls parameter m and the other parameter n. Summands 0 to 3 can be split easily as they only depend on parameters of one of the two components. Summand 4, however, poses a challenge as it depends on both m and n. The two LPEs need to synchronise for the execution of this summand.

Before we define how to split an LPE we need to define projection on vectors. Let $I \subseteq 0, \dots, n$. We define the *I-projection* of $\{d_0, \dots, d_n\}$, denoted by $\langle d_0, \dots, d_n \rangle_{|I}$ as the vector $\langle d_{i_0}, \dots, d_{i_\ell} \rangle$ where ℓ is the largest natural number such that $i_0 < i_\ell < \dots < i_\ell \le n$ and $i_k \in I$ for $0 \le k \le \ell$. For a vector of data expressions and their domains $\vec{d} : \vec{D}$ we denote the projection by $\vec{d}_{|I} : \vec{D}_{|I}$.

Below, we define the notion of a *separation tuple*. Intuitively, a separation tuple defines how to split off part of the process parameters and summands of an LPE. To split an LPE into two parts we need two separation tuples.

Definition 4.2 ([83]). *Let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE. A separation tuple for P is a 6-tuple $(U, K, J, c^U, \alpha^U, \vec{h}^U)$ where $U \subseteq \mathbb{N}$ is a set of parameter indices, $K \subseteq J \subseteq I$ are two sets of summand indices, and c^U, α^U and $\vec{h}^U$ are functions with domain $J \setminus K$ and as codomains the sets of conditions, multi-actions and synchronisation expressions, respectively. A synchronisation expression is just a vector of variables, they represent process parameters that we will synchronise on. We require that for all $i \in (J \setminus K)$ it holds that $\mathsf{FV}(c^U(i)) \cup \mathsf{FV}(\alpha^U(i)) \cup \mathsf{FV}(\vec{h}^U(i)) \subseteq \mathsf{Vars}(\vec{d}) \cup \{e_i\}$, and for all $i \in K$ it holds that $\mathsf{FV}(c_i) \cup \mathsf{FV}(\alpha_i) \cup \mathsf{FV}(\vec{g}_{i|U}) \subseteq \mathsf{Vars}(\vec{d}_{|U}) \cup \{e_i\}$.*

A separation tuple induces an LPE, where $U^c = \mathbb{N} \setminus U$, as follows:

$$\begin{aligned} P_U(\vec{d}_{|U} : \vec{D}_{|U}) = & \mathop{+}_{i \in (J \setminus K)} \sum_{e_i : E_i, \vec{d}_{|U^c} : \vec{D}_{|U^c}} c^U(i) \to \alpha^U(i) | \mathsf{sync}^i_U(\vec{h}^U(i)) \,.\, P_U(\vec{g}_{i|U}) \\ & + \mathop{+}_{i \in K} \sum_{e_i : E_i} c_i \to \alpha_i | \mathsf{tag} \,.\, P_U(\vec{g}_{i|U}) \end{aligned}$$

We assume that action labels sync^i_U, for any $i \in I$, and label tag do not occur in the original LPE.

In Definition 4.2 the set J contains the summand indices of all summands featured in the separation tuple, of which the subset K contains the local summands and subset $J \setminus K$ the summands needing synchronisation.

Example 4.3. *Consider the LPE P from Example 4.1 again. Suppose we decompose P using the separation tuples*

$$(V, \{0,1,2\}, \{0,1,2,4\}, \{4 \mapsto m \approx 1\}, \{4 \mapsto \lbag \mathsf{c} \rbag\}, \{4 \mapsto \langle\rangle\})$$

and

$$(W, \{3\}, \{3,4\}, \{4 \mapsto \neg n\}, \{4 \mapsto \lbag\rbag\}, \{4 \mapsto \langle\rangle\}),$$

where $V = \{0\}$ and $W = \{1\}$ are sets of parameter indices. These separation tuples induce the following LPEs:

$$\begin{aligned}
P_V(m : \texttt{Nat}) &= (m \approx 0) \to \tau|\mathsf{tag} \,.\, P_V(1)\\
&+ (m \approx 1) \to \mathsf{a}|\mathsf{tag} \,.\, P_V(2)\\
&+ (m \approx 1) \to \tau|\mathsf{tag} \,.\, P_V(2)\\
&+ (m \approx 1) \to \mathsf{c}|\mathsf{sync}^4_V \,.\, P_V(2)\\
P_W(n : \texttt{Bool}) &= (\neg n) \to \mathsf{b}|\mathsf{tag} \,.\, P_W(\mathit{true})\\
&+ (\neg n) \to \tau|\mathsf{sync}^4_W \,.\, P_W(\mathit{true})
\end{aligned}$$

Summands 0 to 3 of the original LPE P are local*; they only depend on local process parameters. The* tag *labels are added to prevent local summands from unintended communications in the parallel composition. We revisit the necessity of the* tag *action in Example 4.6. Summand 4 of P, which depends on both m and n is split across P_V and P_W, which will need to synchronise on the* sync^4 *labels. The LTSs induced by the LPEs are given in Example 4.7.*

Note that Definition 4.2 does not specify how to split an LPE into two sensible separation tuples. To *correctly* split an LPE, a number of requirements need to be fulfilled. Two separation tuples form a *cleave* if and only if the requirements listed in [83, Definition 4.6] hold. The definition of the notion of cleave is only needed in proofs that are not repeated in this thesis. We will only restate the relevant theorem, hence there is no need to repeat the definition of cleave here. Separation tuples that are a cleave can be recombined in such a way that the result is strongly bisimilar to the original LPE (Theorem 4.4 below, the main result of [83]). The separation tuples from Example 4.3 constitute a cleave.

Theorem 4.4 states both how a cleaved LPE can be recombined and that the recombined process is strongly bisimilar to the original LPE. A proof of this theorem is provided in [82].

Theorem 4.4 ([83]). *Let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and let $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$ be a cleave for this LPE. Let P_V and P_W be the LPEs induced by the separation tuples. For every closed expression $\vec{\iota} : \vec{D}$ we get the following correspondence between the original process $P(\vec{\iota})$ and the cleaved and recombined process:*

$$\begin{aligned}
P(\vec{\iota}) \mathrel{\underline{\leftrightarrow}} \tau_{\{\mathsf{tag}\}}(\nabla_{\{\underline{\alpha_i} \;\mid\; i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}} \;\mid\; i \in (K^V \cup K^W)\}}(&\\
\tau_{\{\mathsf{sync}^i \;\mid\; i \in I\}}(\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W \to \mathsf{sync}^i \;\mid\; i \in I\}}(P_V(\vec{\iota}|_V) \parallel P_W(\vec{\iota}|_W))))&
\end{aligned}$$

Let us review the purpose of each operator in the recombining process expression. Note that each multi-action in $P_V(\vec{\iota}_{|V})$ and $P_W(\vec{\iota}_{|W})$ includes either a tag or sync label. Also note that due to the multi-action semantics, $P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W})$ can make any combination of steps from $P_V(\vec{\iota}_{|V})$ and $P_W(\vec{\iota}_{|W})$. The outer hide operator simply hides all tag labels. The allow operator is parametrised with an *allow set*, which contains each multi-action α_i of the summands of the original LPE; it also contains multi-actions $\alpha_i|\mathsf{tag}$ if α_i is the action in one of the local transitions. Any other action is blocked. In particular, multi-actions containing two tags or a sync label are blocked. The communication and inner operators ensure that matching sync labels communicate and are subsequently hidden.

In the proofs of Section 4.3 we need that the multiset consisting of a single tag action is always allowed by the allow operator. The allow operator in the process expression of Theorem 4.4 only includes a tag action if there exists an $i \in I$ such that $\alpha^V(i) = \tau$ or $\alpha^W(i) = \tau$. Corollary 4.5 (below) states that we can always add the tag action to the allow set.

Corollary 4.5. *Let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and let $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$ be a cleave for this LPE. Let P_V and P_W be the LPEs induced by the separation tuples. For every closed expression $\vec{\iota} : \vec{D}$ we get the following correspondence between the original process $P(\vec{\iota})$ and the cleaved and recombined process:*

$$
\begin{aligned}
P(\vec{\iota}) \;\underline{\leftrightarrow}\; & \tau_{\{\mathsf{tag}\}}(\nabla_{\{\mathsf{tag}\} \cup \{\underline{\alpha_i} \;\mid\; i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}} \;\mid\; i \in (K^V \cup K^W)\}}(\\
& \qquad \tau_{\{\mathsf{sync}^i \;\mid\; i \in I\}}(\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W \to \mathsf{sync}^i \;\mid\; i \in I\}}(P_V(\vec{\iota}_{|V})) \parallel P_W(\vec{\iota}_{|W}))))
\end{aligned}
$$

Proof. Let $P'(\vec{\iota})$ be an LPE which includes all summands of $P(\vec{\iota})$ and adds a dummy summand $(\mathsf{false}) \to \tau.P(g)$, where g is some arbitrary update expression. Clearly $P(\vec{\iota})$ and $P'(\vec{\iota})$ are strongly bisimilar. Let the added summand be the last summand with index n. Then there is a cleave of $P'(\vec{\iota})$ consisting of the separation tuples $(V \cup \{n\}, K^V \cup \{n\}, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$. By Theorem 4.4 we have the following correspondence:

$$
\begin{aligned}
P'(\vec{\iota}) \;\underline{\leftrightarrow}\; & \tau_{\{\mathsf{tag}\}}(\nabla_{\{\underline{\alpha_i} \;\mid\; i \in (I \cup \{n\})\} \cup \{\underline{\alpha_i|\mathsf{tag}} \;\mid\; i \in (K^V \cup K^W \cup \{n\})\}}(\\
& \tau_{\{\mathsf{sync}^i \;\mid\; i \in (I \cup \{n\})\}}(\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W \to \mathsf{sync}^i \;\mid\; i \in (I \cup \{n\})\}}(P_V(\vec{\iota}_{|V})) \parallel P_W(\vec{\iota}_{|W}))))
\end{aligned}
$$

The allow set can be rewritten to $\{\mathsf{tag}\} \cup \{\underline{\alpha_i} \mid i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}} \mid i \in (K^V \cup K^W)\}$. The extension from I to $I \cup \{n\}$ in the hiding and communication operators does not change the behaviour as $P_{V \cup \{n\}}(\vec{\iota}_{|V \cup \{n\}})$ and $P_W(\vec{\iota}_{|W})$ cannot produce sync^n actions. By transitivity of strong bisimulation we obtain:

$$
\begin{aligned}
P(\vec{\iota}) \;\underline{\leftrightarrow}\; & \tau_{\{\mathsf{tag}\}}(\nabla_{\{\mathsf{tag}\} \cup \{\underline{\alpha_i}|i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}}|i \in (K^V \cup K^W)\}}(\\
& \qquad \tau_{\{\mathsf{sync}^i|i \in I\}}(\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W \to \mathsf{sync}^i|i \in I\}}(P_V(\vec{\iota}_{|V})) \parallel P_W(\vec{\iota}_{|W}))))
\end{aligned}
$$

$\square$

From now on we use the following shorthand for convenience, which is only a well defined process expression in the context of a process and its cleave, which define α, I, V and W.

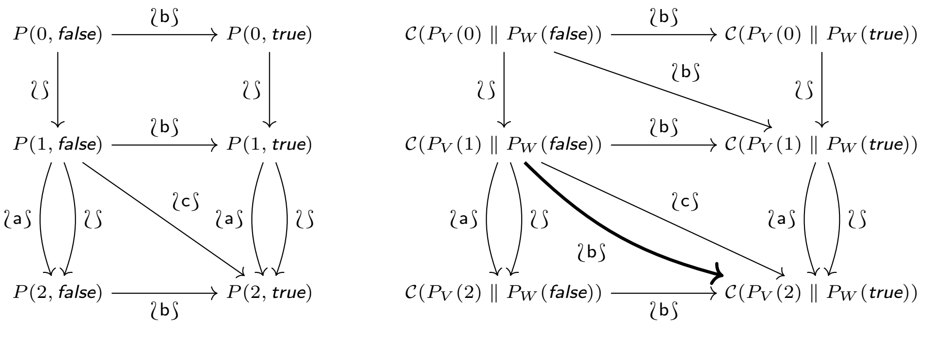
$$\mathcal{C}(P) := \tau_{\{\mathsf{tag}\}}(\nabla_{\{\mathsf{tag}\}\cup\{\underline{\alpha_i} \ \mid \ i\in I\}\cup\{\underline{\alpha_i|\mathsf{tag}} \ \mid \ i\in(K^V\cup K^W)\}}(\\ \tau_{\{\mathsf{sync}^i \ \mid \ i\in I\}}(\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W\to \mathsf{sync}^i \ \mid \ i\in I\}}(P))))$$

Since strong bisimulation is a congruence for the operators used to construct $\mathcal{C}$, processes $P_V(\vec{\iota}_{|V})$ and $P_W(\vec{\iota}_{|W})$ can be replaced by any strongly bisimilar process expression or LTS. The theory is implemented in two tools. Given a set of process parameters the tool `lpscleave` correctly cleaves an LPS into two LPSs. Recall that an LPS consists of both an LPE and a data specification. The tool `lpscombine` applies the operators in the context $\mathcal{C}$ to two LPSs or LTSs.

4.3 Extension to Branching Bisimilarity

The difficulty of supporting branching bisimulation reduction of component LPEs lies in the fact that transitions local to a component are made visible by adding a tag label. These tag labels are necessary to prevent τ-transitions from arbitrarily communicating with other summands, as explained in the following example.

Example 4.6. *Suppose the component LPEs P_V and P_W from Example 4.3 did not have added* tag *actions. As can be seen below, $\mathcal{C}(P_V(1) \parallel P_W(\mathit{false}))$ can perform a* b*-labelled transition to $\mathcal{C}(P_V(2) \parallel P_W(\mathit{true}))$, which cannot be mimicked by $P(1, \mathit{false})$ as the only* b*-labelled transition it can perform leads to a state from which an* a*-labelled transition is enabled. The* b*-labelled transition stems from a synchronisation between the summands $(m \approx 1) \to \tau \,.\, P_V(2)$ and $(\neg n) \to \mathsf{b} \,.\, P_W(\mathit{true})$.*



$$\begin{tikzcd}
P(0,\mathit{false}) \arrow[r, "\lbag \mathsf{b} \rbag"] \arrow[d, "\lbag \rbag"'] & P(0,\mathit{true}) \arrow[d, "\lbag \rbag"'] \\
P(1,\mathit{false}) \arrow[r, "\lbag \mathsf{b} \rbag"] \arrow[d, bend right, "\lbag \mathsf{a} \rbag"'] \arrow[d, bend left, "\lbag \rbag"] \arrow[dr, "\lbag \mathsf{c} \rbag"] & P(1,\mathit{true}) \arrow[d, bend right, "\lbag \mathsf{a} \rbag"'] \arrow[d, bend left, "\lbag \rbag"] \\
P(2,\mathit{false}) \arrow[r, "\lbag \mathsf{b} \rbag"'] & P(2,\mathit{true})
\end{tikzcd}$$

$$\begin{tikzcd}
\mathcal{C}(P_V(0) \parallel P_W(\mathit{false})) \arrow[r, "\lbag \mathsf{b} \rbag"] \arrow[d, "\lbag \rbag"'] \arrow[dr, "\lbag \mathsf{b} \rbag"] & \mathcal{C}(P_V(0) \parallel P_W(\mathit{true})) \arrow[d, "\lbag \rbag"'] \\
\mathcal{C}(P_V(1) \parallel P_W(\mathit{false})) \arrow[r, "\lbag \mathsf{b} \rbag"] \arrow[d, bend right, "\lbag \mathsf{a} \rbag"'] \arrow[d, bend left, "\lbag \rbag"] \arrow[dr, "\lbag \mathsf{c} \rbag"] \arrow[dr, bend right, thick, "\lbag \mathsf{b} \rbag"'] & \mathcal{C}(P_V(1) \parallel P_W(\mathit{true})) \arrow[d, bend right, "\lbag \mathsf{a} \rbag"'] \arrow[d, bend left, "\lbag \rbag"] \\
\mathcal{C}(P_V(2) \parallel P_W(\mathit{false})) \arrow[r, "\lbag \mathsf{b} \rbag"'] & \mathcal{C}(P_V(2) \parallel P_W(\mathit{true}))
\end{tikzcd}$$

Hence, in the absence of tag *labels, the process expressions $P(1, \mathit{false})$ and $\mathcal{C}(P_V(1) \parallel P_W(\mathit{false}))$ are not strongly bisimilar, and therefore also not branching bisimilar. This is problematic for minimisation modulo branching bisimilarity because a side effect of the* tag *actions is that there are no τ-labelled transitions. Hence, we do not get a larger reduction than with strong bisimulation minimisation.*

The solution to minimising modulo branching bisimulation without having communicating τ's in the parallel composition is presented in Theorem 4.11 below. Intuitively, the theorem states that we can replace a component P of a cleaved process by a process P' if P and P' are branching bisimilar *after abstraction from tag-actions*. We illustrate this with an example.

Example 4.7. *Below we give the LTSs of P_V and P_W from Example 4.3.*

$$\begin{tikzcd}
P_V(0) \arrow[r, "\lbag \mathsf{tag} \rbag"] & P_V(1) \arrow[r, "\lbag \mathsf{a}, \mathsf{tag} \rbag"] \arrow[r, bend left, "\lbag \mathsf{tag} \rbag"] \arrow[r, bend right, "\lbag \mathsf{c}, \mathsf{sync}^4_V \rbag"'] & P_V(2)
\end{tikzcd}
\qquad
\begin{tikzcd}
P_W(\mathit{false}) \arrow[r, bend left, "\lbag \mathsf{b}, \mathsf{tag} \rbag"] \arrow[r, bend right, "\lbag \mathsf{sync}^4_W \rbag"'] & P_W(\mathit{true})
\end{tikzcd}$$

Below we show three subsequent transformations on P_V: hiding tag *actions, minimising modulo branching bisimulation and adding a* tag *action to each transition that does not have a* sync *label.*

$$\begin{tikzcd}
s_0 \arrow[r, "\lbag \rbag"] & s_1 \arrow[r, "\lbag \mathsf{a} \rbag"] \arrow[r, bend left, "\lbag \rbag"] \arrow[r, bend right, "\lbag \mathsf{c}, \mathsf{sync}^4_V \rbag"'] & s_2
\end{tikzcd}
\quad
\begin{tikzcd}
t_0 \arrow[r, "\lbag \mathsf{a} \rbag"] \arrow[r, bend left, "\lbag \rbag"] \arrow[r, bend right, "\lbag \mathsf{c}, \mathsf{sync}^4_V \rbag"'] & t_1
\end{tikzcd}
\quad
\begin{tikzcd}
u_0 \arrow[r, "\lbag \mathsf{a}, \mathsf{tag} \rbag"] \arrow[r, bend left, "\lbag \mathsf{tag} \rbag"] \arrow[r, bend right, "\lbag \mathsf{c}, \mathsf{sync}^4_V \rbag"'] & u_1
\end{tikzcd}$$

Below we show the reachable part of the labelled transition system of $\mathcal{C}(u_0 \parallel P_W(\mathit{false}))$. Note that the LTS is branching bisimilar to the original LPE P (see Example 4.6), and also smaller.

$$\begin{tikzcd}
\mathcal{C}(u_0 \parallel P_W(\mathit{false})) \arrow[r, "\lbag \mathsf{b} \rbag"] \arrow[d, bend right, "\lbag \mathsf{a} \rbag"'] \arrow[d, bend left, "\lbag \rbag"] \arrow[dr, "\lbag \mathsf{c} \rbag"] & \mathcal{C}(u_0 \parallel P_W(\mathit{true})) \arrow[d, bend right, "\lbag \mathsf{a} \rbag"'] \arrow[d, bend left, "\lbag \rbag"] \\
\mathcal{C}(u_1 \parallel P_W(\mathit{false})) \arrow[r, "\lbag \mathsf{b} \rbag"'] & \mathcal{C}(u_1 \parallel P_W(\mathit{true}))
\end{tikzcd}$$

Before we present Theorem 4.11, which generalises and formalises the substitution of $P_W(\mathsf{false})$ with u_0 we performed in Example 4.7, we present two definitions and two lemmas to relate transitions possible in process expressions of the form $\mathcal{C}(P \parallel Q)$ and the transitions possible from the parallel components P and Q.

For a process expression P, let $A(P)$ denote the *alphabet* of P, i.e., $A(P) = \{\underline{\omega} \in \Lambda \mid P \to^* P' \xrightarrow{\underline{\omega}} P''\}$.

Definition 4.8. *A process expression P is a* tag/sync *process expression if, and only if, every label in $A(P)$ contains exactly one* tag *or* sync *label:*

$$\forall_{\underline{\omega} \in A(P)}((\textstyle\sum_{a \in \{\mathsf{tag}\} \cup \{\mathsf{sync}^i_V \mid i \in \mathbb{N}, V \subseteq \mathbb{N}\}} \underline{\omega}(a)) = 1).$$

Recall S *from Definition 2.5, which denotes the set of all process expressions. Let* $\mathsf{S}_{ts}(V) \subseteq \mathsf{S}$*, with* $V \subseteq \mathbb{N}$*, denote the set of all* tag/sync *process expressions where each* sync *action has subscript* V*.*

Note that any LPE induced by a cleave is a tag/sync process. The lemma below relates τ-labelled paths from $\tau_{\{\mathsf{tag}\}}(P)$ and $\mathcal{C}(P \parallel Q)$.

Lemma 4.9. *Let there be some LPE and cleave providing the context for* $\mathcal{C}$*. Let* P *and* Q *be* tag/sync *process expressions. Then we have that*

$$\tau_{\{\mathsf{tag}\}}(P) \twoheadrightarrow \tau_{\{\mathsf{tag}\}}(P') \text{ implies } \mathcal{C}(P \parallel Q) \twoheadrightarrow \mathcal{C}(P' \parallel Q)$$

Proof. The proof is by induction on the length of a sequence of τ-transitions from $\tau_{\{\mathsf{tag}\}}(P)$ to $\tau_{\{\mathsf{tag}\}}(P')$. In the base case, when the length of that sequence is 0, there is nothing to prove. In the step case we declare a variable $k \in \mathbb{N}$ denoting the length of the path. The induction hypothesis is stated below.

$$\tau_{\{\mathsf{tag}\}}(P) = \tau_{\{\mathsf{tag}\}}(P_0) \xrightarrow{\Lbag\Rbag} \cdots \xrightarrow{\Lbag\Rbag} \tau_{\{\mathsf{tag}\}}(P_k) \text{ implies } \mathcal{C}(P_0 \parallel Q) \twoheadrightarrow \mathcal{C}(P_k \parallel Q)$$

Now consider a sequence

$$\tau_{\{\mathsf{tag}\}}(P) = \tau_{\{\mathsf{tag}\}}(P_0) \xrightarrow{\Lbag\Rbag} \cdots \xrightarrow{\Lbag\Rbag} \tau_{\{\mathsf{tag}\}}(P_{k+1}) = \tau_{\{\mathsf{tag}\}}(P')$$

of length $k + 1$. By the induction hypothesis $\mathcal{C}(P_0 \parallel Q) \twoheadrightarrow \mathcal{C}(P_k \parallel Q)$. In accordance with the operational rules $P_k \xrightarrow{\Lbag \mathsf{tag}:n \Rbag} P_{k+1}$, with $n \in \mathbb{N}$. Note that by the assumption that P is a tag/sync-process (and P_k as well, since it is reachable from P) we have that $n = 1$. Since $\Lbag\mathsf{tag}\Rbag$ is in the allow set of $\mathcal{C}$, it follows that $\mathcal{C}(P_k \parallel Q) \xrightarrow{\Lbag\Rbag} \mathcal{C}(P_{k+1} \parallel Q)$. Hence $\mathcal{C}(P_0 \parallel Q) \twoheadrightarrow \mathcal{C}(P_{k+1} \parallel Q)$. □

The next lemma relates transitions of $\mathcal{C}(P \parallel Q)$ and P and Q.

Lemma 4.10. *Let* $L(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, L(\vec{g}_i)$ *be an LPE and let* $(V, K^V, J^V, c^V, \alpha^V, h^V)$ *and* $(W, K^W, J^W, c^W, \alpha^W, h^W)$ *be a cleave. Let* $P \in \mathsf{S}_{ts}(V)$ *and* $Q \in \mathsf{S}_{ts}(W)$ *be* tag/sync *process expressions. For every transition* $\mathcal{C}(P \parallel Q) \xrightarrow{\omega} \mathcal{C}(P' \parallel Q')$ *we have that either*

- $P \xrightarrow{\omega + \Lbag\mathsf{tag}\Rbag} P'$ *and* $Q = Q'$*;*
- $Q \xrightarrow{\omega + \Lbag\mathsf{tag}\Rbag} Q'$ *and* $P = P'$ *or*
- $P \xrightarrow{\omega_1 + \Lbag \mathsf{sync}^i_V(\vec{\boldsymbol{d}}) \Rbag} P', Q \xrightarrow{\omega_2 + \Lbag \mathsf{sync}^i_W(\vec{\boldsymbol{d}}) \Rbag} Q', \omega_1 + \omega_2 = \omega.$

Proof. Suppose $\mathcal{C}(P \parallel Q) \xrightarrow{\omega} \mathcal{C}(P' \parallel Q')$. Unfolding the shorthand $\mathcal{C}$:

$$\begin{aligned}
&\tau_{\{\mathsf{tag}\}}(\nabla_{\{\mathsf{tag}\} \cup \{\underline{\alpha_i} \,\mid\, i \in I\} \cup \{\underline{\alpha_i | \mathsf{tag}} \,\mid\, i \in (K^V \cup K^W)\}}(\tau_{\{\mathsf{sync}^i \,\mid\, i \in I\}}(\\
&\Gamma_{\{\mathsf{sync}^i_V | \mathsf{sync}^i_W \to \mathsf{sync}^i \,\mid\, i \in I\}}(P \parallel Q)))) \xrightarrow{\omega} \tau_{\{\mathsf{tag}\}}(\\
&\nabla_{\{\mathsf{tag}\} \cup \{\underline{\alpha_i} \,\mid\, i \in I\} \cup \{\underline{\alpha_i | \mathsf{tag}} \,\mid\, i \in (K^V \cup K^W)\}}(\\
&\qquad\qquad\qquad \tau_{\{\mathsf{sync}^i \,\mid\, i \in I\}}(\Gamma_{\{\mathsf{sync}^i_V | \mathsf{sync}^i_W \to \mathsf{sync}^i \,\mid\, i \in I\}}(P' \parallel Q'))))
\end{aligned}$$

The last rule applied in the derivation of this transition must be Hide, with the following premise, for some $n \in \mathbb{N}$:

$$\begin{aligned}&\nabla_{\{\mathsf{tag}\}\cup\{\underline{\alpha_i} \mid i\in I\}\cup\{\underline{\alpha_i|\mathsf{tag}} \mid i\in(K^V\cup K^W)\}}(\tau_{\{\mathsf{sync}^i \mid i\in I\}}(\\ &\quad\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W\to\mathsf{sync}^i \mid i\in I\}}(P \parallel Q))) \xrightarrow{\omega+\wr\mathsf{tag}:n\int} \\ &\qquad\nabla_{\{\mathsf{tag}\}\cup\{\underline{\alpha_i} \mid i\in I\}\cup\{\underline{\alpha_i|\mathsf{tag}} \mid i\in(K^V\cup K^W)\}}(\\ &\tau_{\{\mathsf{sync}^i \mid i\in I\}}(\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W\to\mathsf{sync}^i \mid i\in I\}}(P' \parallel Q')))\end{aligned}$$

The last rule applied in the derivation of this transition must be Allow, with the following premise, and the restriction that $\omega_1 \in \{[\![\alpha]\!] \mid \alpha \in \{\mathsf{tag}\} \cup \{\underline{\alpha_i} \mid i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}} \mid i \in (K^V \cup K^W)\}\}$:

$$\begin{aligned}&\tau_{\{\mathsf{sync}^i \mid i\in I\}}(\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W\to\mathsf{sync}^i \mid i\in I\}}(P \parallel Q)) \xrightarrow{\omega_1}\\ &\qquad\qquad\tau_{\{\mathsf{sync}^i \mid i\in I\}}(\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W\to\mathsf{sync}^i \mid i\in I\}}(P' \parallel Q'))\end{aligned}$$

The last rule applied in the derivation of this transition must be Hide, with the following premise.

$$\Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W\to\mathsf{sync}^i \mid i\in I\}}(P \parallel Q) \xrightarrow{\omega_1+\omega_2} \Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W\to\mathsf{sync}^i \mid i\in I\}}(P' \parallel Q')$$

There is the restriction that ω_2 consists solely of sync actions, i.e. for all a such that $\underline{\omega_2}(a) > 0$ we have that $a \in \{\mathsf{sync}^i \mid i \in \mathbb{N}\}$.

The last rule applied in the derivation of this transition must be Com, with the following premise.

$$P \parallel Q \xrightarrow{\omega_1+\omega_3+\omega_4} P' \parallel Q'$$

It is required that ω_3 and ω_4 consist of complementing sync actions:

$$\begin{gathered}\text{For all } a \text{ such that } \underline{\omega_3}(a) > 0 : a \in \{\mathsf{sync}^i_V \mid i \in \mathbb{N}\}\\ \text{For all } a \text{ such that } \underline{\omega_4}(a) > 0 : a \in \{\mathsf{sync}^i_W \mid i \in \mathbb{N}\}\\ \text{For all } i \in \mathbb{N} \text{ and } \vec{\boldsymbol{d}} \in \mathbb{D}_{\mathsf{sync}^i_V} : \omega_3(\mathsf{sync}^i_V(\vec{\boldsymbol{d}})) = \omega_4(\mathsf{sync}^i_W(\vec{\boldsymbol{d}}))\end{gathered}$$

Note that $P \in \mathsf{S}_{ts}(V)$ and $Q \in \mathsf{S}_{ts}(W)$ are tag/sync process expressions. Hence, for any derivation with as conclusion $P \xrightarrow{\omega'} P'$ or $Q \xrightarrow{\omega'} Q'$ we have that ω' must contain either one tag action or one sync action. We can distinguish three cases for the rule applied in the derivation of $P \parallel Q \xrightarrow{\omega_1+\omega_3+\omega_4} P' \parallel Q'$:

- Par: due to the restrictions on the multiset $\omega_1 + \omega_3 + \omega_4$ and the fact that P and Q are tag/sync process expressions we can exclude a number of possibilities: P and Q cannot both contribute a tag action (since ω_1 contains at most one) and it cannot be the case that either P or Q contributes a tag action and the other a sync action (as syncs must come in matching pairs). The only possibility is that P and Q contribute matching sync actions and ω_1 is split in some way: $P \xrightarrow{\omega_1^1+\wr\mathsf{sync}^i_V(\vec{\boldsymbol{d}})\int} P', Q \xrightarrow{\omega_1^2+\wr\mathsf{sync}^i_W(\vec{\boldsymbol{d}})\int} Q', \omega_1^1 + \omega_1^2 = \omega$.

- ParL: Only P contributes to the derivation. Since P cannot make a step with matching sync actions it must make a step with a single tag action: $P \xrightarrow{\omega + \wr \mathsf{tag} \wr} P'$ and $Q = Q'$.
- ParR: Only Q contributes to the derivation. Since Q cannot make a step with matching sync actions it must make a step with a single tag action: $Q \xrightarrow{\omega + \wr \mathsf{tag} \wr} Q'$ and $P = P'$.

$\square$

We are now ready to prove the main contribution of this chapter. The theorem states that, in the context $\mathcal{C}(P \parallel Q)$, we can swap out component LPE P with some process expression L if $\tau_{\{\mathsf{tag}\}}(P) \mathrel{\underline{\leftrightarrow}_b} \tau_{\{\mathsf{tag}\}}(L)$.

Theorem 4.11. *Let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and let $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$ be a cleave. Let $\vec{\iota} : \vec{D}$ be an arbitrary closed expression. Let $L \in \mathsf{S}_{ts}(V)$ and $R \in \mathsf{S}_{ts}(W)$ be* tag/sync *process expressions such that $\tau_{\{\mathsf{tag}\}}(P_V(\vec{\iota}_{|V})) \mathrel{\underline{\leftrightarrow}_b} \tau_{\{\mathsf{tag}\}}(L)$ and $\tau_{\{\mathsf{tag}\}}(P_W(\vec{\iota}_{|W})) \mathrel{\underline{\leftrightarrow}_b} \tau_{\{\mathsf{tag}\}}(R)$. We then have*

$$P(\vec{\iota}) \mathrel{\underline{\leftrightarrow}_b} \mathcal{C}(L \parallel R).$$

Proof. By Theorem 4.4, we have: $P(\vec{\iota}) \mathrel{\underline{\leftrightarrow}} \mathcal{C}(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}))$ and therefore $P(\vec{\iota}) \mathrel{\underline{\leftrightarrow}_b} \mathcal{C}(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}))$. It therefore suffices to show that $\mathcal{C}(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W})) \mathrel{\underline{\leftrightarrow}_b} \mathcal{C}(L \parallel R)$, which we prove by showing that

$$\begin{aligned}\mathcal{R} = \{(\mathcal{C}(P \parallel Q), \mathcal{C}(P' \parallel Q')) \mid {} & P, P' \in \mathsf{S}_{ts}(V) \wedge Q, Q' \in \mathsf{S}_{ts}(W) \text{ tag/sync} \\ & \text{process expressions s.t. } \tau_{\{\mathsf{tag}\}}(P) \mathrel{\underline{\leftrightarrow}_b} \tau_{\{\mathsf{tag}\}}(P') \wedge \tau_{\{\mathsf{tag}\}}(Q) \mathrel{\underline{\leftrightarrow}_b} \tau_{\{\mathsf{tag}\}}(Q')\}\end{aligned}$$

is a branching bisimulation relation. Note that $P_V(\vec{\iota}_{|V})$ and $P_W(\vec{\iota}_{|W})$ are tag/sync process expressions. To check the transfer conditions of the pairs in $\mathcal{R}$ we pick arbitrary process expressions P, P', Q and Q' meeting the conditions of the definition of $\mathcal{R}$.

Suppose $\mathcal{C}(P \parallel Q) \xrightarrow{\omega} \mathcal{C}(P'' \parallel Q'')$. By Lemma 4.10 there are three options for the contributions of P and Q.

- $P \xrightarrow{\omega + \wr \mathsf{tag} \wr} P''$ and $Q = Q'$. Since $\tau_{\{\mathsf{tag}\}}(P) \mathrel{\underline{\leftrightarrow}_b} \tau_{\{\mathsf{tag}\}}(P')$, there exist S and S' such that

  $$\tau_{\{\mathsf{tag}\}}(P') \twoheadrightarrow \tau_{\{\mathsf{tag}\}}(S) \xrightarrow{(\omega)} \tau_{\{\mathsf{tag}\}}(S')$$

  with both $\tau_{\{\mathsf{tag}\}}(P) \mathrel{\underline{\leftrightarrow}_b} \tau_{\{\mathsf{tag}\}}(S)$ and $\tau_{\{\mathsf{tag}\}}(P'') \mathrel{\underline{\leftrightarrow}_b} \tau_{\{\mathsf{tag}\}}(S')$.

  By Lemma 4.9 we have that $\mathcal{C}(P' \parallel Q') \twoheadrightarrow \mathcal{C}(S \parallel Q')$. Since P' is a tag/sync process, so is S; hence a single tag action is hidden by the application of the rule Hide in the derivation of the transition $\tau_{\{\mathsf{tag}\}}(S) \xrightarrow{(\omega)} \tau_{\{\mathsf{tag}\}}(S')$. Therefore $S \xrightarrow{(\omega + \wr \mathsf{tag} \wr)} S'$. Hence $\mathcal{C}(S \parallel Q') \xrightarrow{(\omega)} \mathcal{C}(S' \parallel Q')$. Due to our definition of $\mathcal{R}$, $\tau_{\{\mathsf{tag}\}}(P) \underline{\leftrightarrow}_b \tau_{\{\mathsf{tag}\}}(S)$ and $\tau_{\{\mathsf{tag}\}}(P'') \underline{\leftrightarrow}_b \tau_{\{\mathsf{tag}\}}(S')$ we have that

$$(\mathcal{C}(P \parallel Q), \mathcal{C}(S \parallel Q')) \in \mathcal{R} \text{ and } (\mathcal{C}(P'' \parallel Q), \mathcal{C}(S' \parallel Q')) \in \mathcal{R}.$$

The transfer conditions of Definition 2.10 are hereby satisfied.

- $Q \xrightarrow{\omega + \wr \mathsf{tag} \wr} Q'$ and $P = P'$: completely symmetric to the first case.
- $P \xrightarrow{\omega_1 + \wr \mathsf{sync}^i_V(\vec{d}) \wr} P'', Q \xrightarrow{\omega_2 + \wr \mathsf{sync}^i_W(\vec{d}) \wr} Q''$, and $\omega = \omega_1 + \omega_2$. Since we have that $\tau_{\{\mathsf{tag}\}}(P) \underline{\leftrightarrow}_b \tau_{\{\mathsf{tag}\}}(P')$ there must exist S and S' such that

$$\tau_{\{\mathsf{tag}\}}(P') \twoheadrightarrow \tau_{\{\mathsf{tag}\}}(S) \xrightarrow{\omega_1 + \wr \mathsf{sync}^i_V(\vec{d}) \wr} \tau_{\{\mathsf{tag}\}}(S')$$

with $\tau_{\{\mathsf{tag}\}}(P) \underline{\leftrightarrow}_b \tau_{\{\mathsf{tag}\}}(S)$ and $\tau_{\{\mathsf{tag}\}}(P'') \underline{\leftrightarrow}_b \tau_{\{\mathsf{tag}\}}(S')$. Furthermore, since $\tau_{\{\mathsf{tag}\}}(Q) \underline{\leftrightarrow}_b \tau_{\{\mathsf{tag}\}}(Q')$ there must exist T and T' such that

$$\tau_{\{\mathsf{tag}\}}(Q') \twoheadrightarrow \tau_{\{\mathsf{tag}\}}(T) \xrightarrow{\omega_2 + \wr \mathsf{sync}^i_W(\vec{d}) \wr} \tau_{\{\mathsf{tag}\}}(T')$$

with $\tau_{\{\mathsf{tag}\}}(Q) \underline{\leftrightarrow}_b \tau_{\{\mathsf{tag}\}}(T)$ and $\tau_{\{\mathsf{tag}\}}(Q'') \underline{\leftrightarrow}_b \tau_{\{\mathsf{tag}\}}(T')$.

By Lemma 4.9 we have that $\mathcal{C}(P' \parallel Q') \twoheadrightarrow \mathcal{C}(S \parallel T)$. Since $S \xrightarrow{\omega_1 + \wr \mathsf{sync}^i_V(\vec{d}) \wr} S'$ and $T \xrightarrow{\omega_2 + \wr \mathsf{sync}^i_W(\vec{d}) \wr} T'$ (the restrictions on the alphabet of P' and Q' prohibit extra tags in the multi-actions), we have that $\mathcal{C}(S \parallel T) \xrightarrow{\omega_1 + \omega_2} \mathcal{C}(S' \parallel T')$. Due to the definition of $\mathcal{R}$ we also have that

$$(\mathcal{C}(P \parallel Q), \mathcal{C}(S \parallel T)) \in \mathcal{R} \text{ and } (\mathcal{C}(P'' \parallel Q''), \mathcal{C}(S' \parallel T')) \in \mathcal{R}.$$

The transfer conditions of Definition 2.10 are satisfied.

□

4.4 Minimisation

Theorem 4.11 establishes sufficient conditions for correct replacement of a component; we are interested in replacing a component by one that is minimal with respect to branching bisimilarity. In this section we will formalise the process shown in Example 4.7, in which we hide the tag actions, minimise the LTS and reintroduce tag actions for all transitions that do not have a sync label.

Definition 4.12. *We extend the process algebra of Definition 2.5 with a tagging operator $T_H(P)$, where $H \subseteq \Lambda$ is a non-empty finite set of action labels. Let the extended grammar be denoted by S_t and let S_t denote the set of all process expressions that can be constructed from the grammar.*

Definition 4.13. *We presuppose some LTS $(St, \Omega, \to_1)$. We associate an LTS $(\mathsf{S}_t, \Omega, \to)$ to expressions of S_t, where S_t is parametrised with constants St. The relation $\to$ is the least relation including the transition relation $\to_1$, the rules of Definition 2.8 and the rules below. For any $\omega \in \Omega$, expressions $P, P' \in \mathsf{S}_t$ and $H \subseteq \Lambda$:*

$$\frac{P \xrightarrow{\omega} P'}{T_H(P) \xrightarrow{\omega} T_H(P')} \ \underline{\omega} \cap H \neq \emptyset \qquad\qquad \frac{P \xrightarrow{\omega} P'}{T_H(P) \xrightarrow{\omega + \wr\mathsf{tag}\wr} T_H(P')} \ \underline{\omega} \cap H = \emptyset$$

When the tagging operator is enclosed by a hiding operator that hides tag actions, the tagging operator has no effect, see Lemma 4.14.

Lemma 4.14. *For all $H \subseteq \Lambda$ and $P \in \mathsf{S}$: $\tau_{\{\mathsf{tag}\}}(P) \leftrightarrow \tau_{\{\mathsf{tag}\}}(T_H(P))$.*

Proof. Trivial. For any step $P \xrightarrow{\omega} P'$, both $\tau_{\{\mathsf{tag}\}}(P) \xrightarrow{\omega - \wr\mathsf{tag}\wr} \tau_{\{\mathsf{tag}\}}(P')$ and $\tau_{\{\mathsf{tag}\}}(T_H(P)) \xrightarrow{\omega - \wr\mathsf{tag}\wr} \tau_{\{\mathsf{tag}\}}(T_H(P'))$. □

Theorem 4.15 (below) states that the procedure of hiding, minimising and tagging yields components that can be composed in context $\mathcal{C}$ whilst preserving branching bisimilarity. Processes L and R can be chosen to be minimal with respect to branching bisimilarity.

Theorem 4.15. *Let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and let $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$ be a cleave. Let $\vec{\iota} : \vec{D}$ be an arbitrary closed expression. Let L and R be process expressions such that they are branching bisimilar to $\tau_{\{\mathsf{tag}\}}(P_V(\vec{\iota}_{|V}))$ and $\tau_{\{\mathsf{tag}\}}(P_W(\vec{\iota}_{|W}))$, respectively. Let $H = \{\mathsf{sync}_V^i \mid i \in \mathbb{N}\} \cup \{\mathsf{sync}_W^i \mid i \in \mathbb{N}\}$.*

$$P(\iota) \leftrightarrow_b \mathcal{C}(T_H(L) \parallel T_H(R))$$

Proof. Note that $T_H(L)$ and $T_H(R)$ are $\mathsf{tag}/\mathsf{sync}$ processes: both have either a tag or sync label in every reachable transition. By Lemma 4.14

$$\tau_{\{\mathsf{tag}\}}(T_H(L)) \leftrightarrow \tau_{\{\mathsf{tag}\}}(L) \text{ and } \tau_{\{\mathsf{tag}\}}(T_H(R)) \leftrightarrow \tau_{\{\mathsf{tag}\}}(R).$$

As $L \leftrightarrow_b \tau_{\{\mathsf{tag}\}}(P_V(\vec{\iota}_{|V}))$ and $R \leftrightarrow_b \tau_{\{\mathsf{tag}\}}(P_W(\vec{\iota}_{|W}))$ we have that

$$\tau_{\{\mathsf{tag}\}}(T_H(L)) \leftrightarrow_b \tau_{\{\mathsf{tag}\}}(P_V(\vec{\iota}_{|V})) \text{ and } \tau_{\{\mathsf{tag}\}}(T_H(R)) \leftrightarrow_b \tau_{\{\mathsf{tag}\}}(P_W(\vec{\iota}_{|W})).$$

The requirements of Theorem 4.11 are satisfied, finishing the proof. □

The tool `lpscombine` has been extended with the option to automatically add a tag label to every transition that does not contain a sync label in the LTSs it is combining.

4.5 Experimental Results

In this section we report on the effect of the state space reduction techniques discussed in this chapter on several FormaSig models. Besides presenting statistics on the size of the state space we also compare compositional minimisation to another technique that aims to support model checking of large models: symbolic model checking. The models used for the experiments in this section, along with instructions on how to replicate our results, can be found in the Zenodo repository [12].

4.5.1 Pulse Packs

With pulse packs, the state space of a model with just the S_SCI_LC_SR block of the EULYNX SCI-LC interface contains 46,347 states. Without pulse packs the state space contains at least $5 \cdot 10^8$ states. We were not able to explore the entire state space within a day. The symbolic tool `lpsreach` also was not able to compute the size of the state space as it needs parallelism to obtain performance benefits with the symbolic techniques. The state space becomes so large because for the component in isolation all its input data parameter ports are open to the environment and can be set to any value at any time. When the component is put into composition with other components its inputs are restricted by the behaviour of the connected components. Pulse packs and resetting parameter ports are therefore essential to apply compositional minimisation as we need to be able to efficiently compute the state space of a single component.

4.5.2 Compositional Minimisation

Component	**#states**	**#states (modulo $\underline{\leftrightarrow}$)**	**#states (modulo $\underline{\leftrightarrow}_b$)**
scpPrim	150	76	31
scpSec	174	73	24
prim	259	104	38
sec	636	137	37
est	561	212	77
smi	1788	166	51
flc	20,127	10,052	2,373
slc	17,378	11,011	2,058
functions	432,954	64,639	19,940

Table 4.1: Metrics on the state spaces of components of the full SCI-LC interface.

In the mCRL2 model resulting from the translation of the EULYNX interface, the state machines can be recognised as components, and thus there is a natural partitioning of the parameters resulting in a suitable cleave. Branching bisimulation minimisation is effective on these components (see Table 4.1). The reason is that the transitions related to transition selection and execution (`selectMultiStep`, `executeStep`, `executeBehaviour` and `discardEvent`) can be hidden. Moreover, these transitions are local to the components so they are not labelled with a `sync` label, creating potential for minimisation if the transition is inert.

We ran our experiments on the generic part of the interfaces, on the interface-specific parts of the point interface (SCI-P) and the level crossing interface (SCI-LC), on the combined generic and specific parts of the level crossing interface (full SCI-LC) and on an alternative version of the level crossing interface used in some countries (SCI-LX).

The tools `lpscleave` and `lpscombine` always work on two components. To support multiple components we cleave recursively, in each step splitting off one component; Combining is performed in reverse order. The workflow is depicted in Figure 4.1.

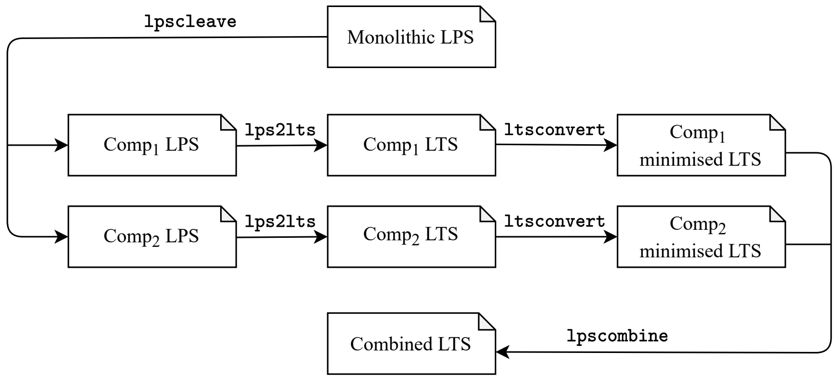


Figure 4.1: Workflow for compositional minimisation.

Table 4.2 shows how much the state space is reduced using branching bisimulation minimisation, the number of states for the monolithic models are computed using the symbolic exploration tool `lpsreach`, which is part of the mCRL2 toolset. The reduction achieved by intermediate minimisation is 3 to 6 orders of magnitude and scales with the number of components. For the full SCI-LC model, neither `lpsreach` nor the compositional approach is able to explore it within a day. For most models, intermediate strong bisimulation minimisation was not sufficient to make state space generation tractable. We were only able to obtain the state space of the generic model, which consists of 1.09696×10^9 states.

Table 4.3 shows how much time each step of the compositional approach takes. Linearisation and cleaving are quick operations. State space exploration and minimisation of components can be performed in parallel, making it independent of

Model	**#states monolithic**	**#states compositional**	**Reduction factor**	**#components**
generic	4.9023×10^{10}	1.1050×10^{6}	4.4363×10^{4}	6
SCI-LC specific	2.7121×10^{12}	2.5961×10^{9}	1.0556×10^{3}	3
SCI-LX	1.3353×10^{12}	1.2540×10^{6}	1.0649×10^{6}	6
SCI-P specific	1.2129×10^{12}	1.4563×10^{7}	8.3294×10^{4}	3
full SCI-LC	$> 3 \times 10^{18}$	unknown	unknown	9

Table 4.2: Metrics reduction factor by intermediate branching bisimulation minimisation.

the number of components. The time to combine the LTSs depends on the size of the final LTS. Any LTS small enough for model checking (less than a billion states) can be generated within approximately a day. All benchmarks were performed on a machine equipped with four 12-core Intel Xeon Gold 6136 CPUs and 3TB of RAM.

Model	**Linearisation + cleaving**	**Generating + minimising LTSs**	**Combining LTSs**	**Total**
generic	5s	166s	65s	236s
SCI-LC specific	9s	2,883s	93,744s	96,636s
SCI-LX	7s	181s	273s	460s
SCI-P specific	3s	557s	369s	929s

Table 4.3: Metrics state space exploration compositional approach.

Requirement Verification. In an earlier case study of the EULYNX point interface [21] we hit the limits of the model checking capabilities of mCRL2. For the SCI-P specific model we could not generate the entire state space. mCRL2 offers both symbolic model checking [81] and explicit state model checking. The symbolic tools do not explore the state space explicitly; they therefore typically provide much better scalability. A benefit of the explicit tools is that they can provide counterexamples [30, 116]. For the point interface we used the symbolic model checking tools but they were still not able to verify any of the requirements.

Below we review whether compositional minimisation combined with the explicit state model-checking tools offer better performance than the symbolic model-checking tools. In mCRL2, a formula is verified by constructing and subsequently solving a PBES from the combination of the formula and the LPS or LTS. Table 4.4 shows the time it takes to verify the formula $[true*]\langle true\rangle true$, expressing that every reachable state has some outgoing transition; which we can check for all models since it does not depend on specific action labels. The symbolic tool `pbessolvesymbolic` is applied to a PBES derived from the LPS. The explicit tools are applied to a PBES obtained from the LTS, which was obtained from the compositional approach. Statistics of the explicit tool are provided with and without counterexample generation enabled. The tools add PBES equations to be able to extract a counterexample at the end, which slows down the solver significantly, as can be seen.

Model	**symbolic**	**explicit**	**explicit + counter**
generic	>7 days	112s	3,169s
SCI-LC specific	>7 days	>7 days	>7 days
SCI-LX	>7 days	116s	4,210s
SCI-P specific	>7 days	1,854s	69,621s

Table 4.4: Metrics on the time needed for verification using various approaches.

For more involved properties our experience is similar; The explicit verification tool paired with compositional minimisation outperforms the symbolic model checking tool, with the extra benefit of providing counterexamples. The explicit tools cannot be used without compositional minimisation due to the size of the state spaces (see Table 4.2).

Applied on the examples considered in [83] to illustrate the effect of decompositional strong bisimulation minimisation, the extension to branching bisimulation minimisation presented here does not lead to significantly smaller components. The reason is that the components in those examples have hardly any internal behaviour.

4.5.3 Restricting the Environment

In Section 4.1.3 we proposed several ways to limit the state space explosion due to communication with the environment. One way is to make communication synchronous, any message from the environment is immediately used to trigger a transition, or discarded. Another way is to let the environment only send a message when there is no event already in the queue of one of the components, i.e. the environment waits until the system is idle. Table 4.5 shows the results of these measures on the state space of the SCI-LX level crossing interface. Note that the model is not exactly the same as in previous experiments. This experiment was performed later in the project with a newer version of the toolchain.

Model	**#states monolithic**	**#states compositional**
SCI-LX, no restrictions	1.2876×10^{10}	7.1980×10^{7}
SCI-LX + synchronous environment	2.6021×10^{9}	3.9482×10^{7}
SCI-LX + wait until idle	3.7755×10^{7}	2.4280×10^{6}
SCI-LX + synchronous environment + wait until idle	3.7755×10^{7}	2.4280×10^{6}

Table 4.5: Metrics on the effect of environment restrictions on the state space. For each model variant compositional branching bisimulation minimisation was used.

Synchronous communication with the environment yields a small reduction of the state space, possibly because the SCI-LX model does not have many ports open to the environment. Making the environment wait until the system is idle does yield a sizeable reduction of factor 30. Combining the two does not lead to a further reduction.

4.6 Conclusion

We have shown that we are able to reduce the state space induced by the formalised EULYNX models by several orders of magnitude. The extension of `lpscombine` should also be effective for other models with similar features. When only communications between components are renamed to τ, component-based minimisation does not reduce the state space. In our models three factors come together enabling a large reduction in the state space:

1. All τ transitions are local to a single component;
2. Most τ transitions are inert;
3. It is difficult to adapt the model to prevent inert τ transitions.

It is likely that other behavioural equivalences (such as weak bisimulation and divergence preserving branching bisimulation) can also be used to safely minimise the LTS of components in which `tag` actions have been hidden. Extending the theory further would entail repeating the proof of Theorem 4.11 for other behavioural equivalences, which is left as future work.

Chapter 5

Case Studies

If the highest aim of a captain were to preserve his ship, he would keep it in port forever.

Thomas Aquinas

In the preceding chapters we presented a formal verification toolchain for SysML models. This is not just an academic matter; there are concrete EULYNX models to analyse. In fact, many (academic) advancements regarding scalability stem from attempts to verify requirements for EULYNX models. In this chapter we go through a number of case studies considering different EULYNX interfaces. This chapter serves multiple purposes. Firstly, it demonstrates the strength of our toolchain, and its limitations. Secondly, it collects all our findings regarding shortcomings in the EULYNX standard. Lastly, and perhaps most importantly, it shows the lessons learned during verification: how we refined requirements through an iterative process and how the subtleties of SysML semantics can introduce issues.

Within FormaSig we restricted ourselves to the following interfaces: SCI-P (point), SCI-LX (the variant of level crossing used by DB) and SCI-LC (the variant of level crossing used by ProRail). The common prefix 'SCI' denotes 'Secure Control Interface'. EULYNX also defines maintenance (SMI) and diagnostics (SDI) interfaces, which are outside the scope of FormaSig.

The case studies have been performed over an extensive time period, using various versions of the toolchain. In writing this thesis the case studies were revisited, using the latest version of the toolchain. With the exception of the analysis of the generic and point interface (see [20, 21]) the material in this chapter has not been published elsewhere.

For documentation and replication purposes the mCRL2 models and μ-calculus formulas used in the case studies can all be found in a Zenodo repository [12]. This repository also includes instructions on how to run the verifications.

Interface name	Field element	Document	Section
Generic	All	Eu.Doc.20, version 3.1	5.4
SCI-P	Point/Switch	Eu.Doc.36, version 2.7	5.5
SCI-LX	Level crossing	Eu.Doc.111, version 1.0	5.3
SCI-LC	Level crossing	Eu.Doc.108, version 1.0	5.6

Table 5.1: Overview of the case studies. The document column indicates from which EULYNX document the SysML diagrams are sourced.

The remainder of this chapter is structured as follows. In Section 5.1 we present the general structure of (the SysML models of) EULYNX interfaces. Section 5.2 explains the common verification approach consisting of the method of requirements elicitation and the toolchain used. Table 5.1 shows which EULYNX interface is treated in which section. In Section 5.7 we discuss the limitations of our analysis. Section 5.8 concludes the chapter with a retrospective of the lessons learned.

5.1 Structure EULYNX Interfaces and Models

The messages exchanged over the network, the logic of the object controller and the interfacing logic of the interlocking constitute the EULYNX standard (see Figure 5.1). Note that the electrical connections between the object controller and the actual hardware is left unspecified.

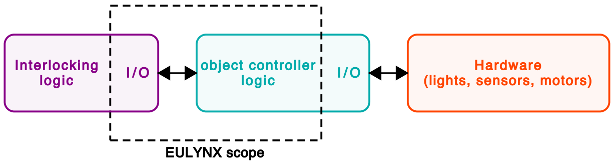


Figure 5.1: Scope of EULYNX interfaces. The connection between the interlocking and the object controller is packet-based and runs over an IP network. The connection between the object controller and the field element is not specified by EULYNX but typically electrical.

The communication between the interlocking and the object controller is distributed over three layers (see Figure 5.2). The first layer manages a channel that follows the *RaSTA protocol*. RaSTA is a safety-focused rail network protocol, described in a DIN standard [114]. The second layer extends the first layer with functionality for identifying the version of EULYNX running on the device on the other side. It also reports general failure modes, such as power loss. The first and second layer are shared by all interfaces. There are two variants of the generic layer namely a variant for subsystems (i.e. field elements) and a variant for adjacent systems (e.g. for the interlocking-interlocking interface). The third layer is specific to the interface at hand (point, light signal, etcetera).

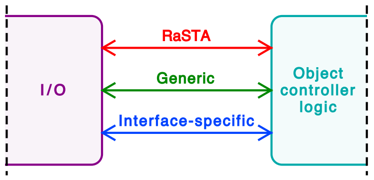


Figure 5.2: Layers of communication between the interlocking and the object controller.

Each layer of a EULYNX interface consists of a number of components which are represented as blocks in the IBDs. See Figure 5.3 for an example of a component decomposition. Some blocks model behaviour of the interlocking whilst other blocks model behaviour of the object controller. The RaSTA protocol is actually not modelled explicitly in EULYNX. The interface to RaSTA is modelled but there are no components modelling the behaviour of RaSTA. It is part of the environment with which EULYNX may interact. We modelled the behaviour of RaSTA ourselves with blocks S_SCI_SCP_Prim_SR and F_SCI_SCP_Sec_SR, to be able to analyse the interaction between RaSTA and EULYNX. Our modelling of RaSTA can be found in the Zenodo repository [12].

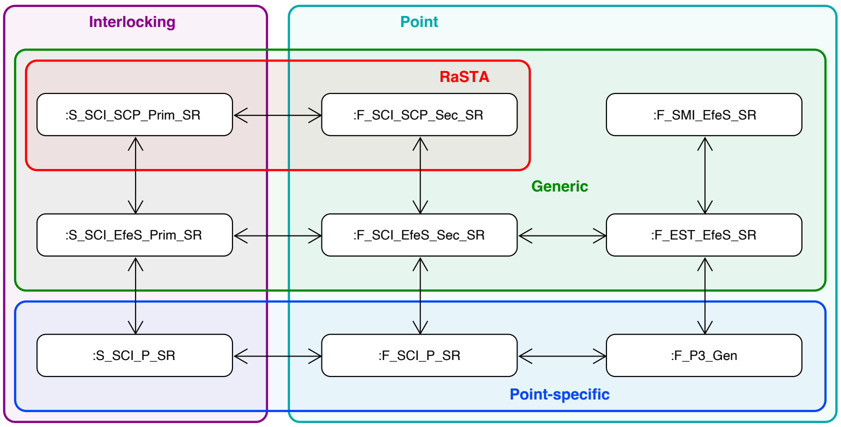


Figure 5.3: Structure of the components of the EULYNX Point interface. Arrows indicate communication channels between components.

5.2 Common Approach

Before we move on to the specifics of each case study we discuss our approach regarding the strategy to requirement elicitation and the strategy to reduce the state space.

5.2.1 Requirements Elicitation

The EULYNX standard specifies, through sequence diagrams, several scenarios for each interface. The state machines of the interface should at least admit the correct execution of these scenarios. They can be interpreted as requirements. However, to assess the quality of the EULYNX standard, we need to verify stronger requirements: rather than one scenario, we need to verify that properties hold along all execution paths of the system. The system should *never* be allowed to perform behaviour that leads to hazardous situations. Eliciting such requirements is an explicit concern of the FormaSig project, as good requirements are essential to assess the quality of EULYNX specifications.

A challenge in formulating pertinent requirements is that the prerequisite knowledge is, currently, split between the academic partners of FormaSig and the infrastructure managers: only the former possess the skills to formulate formal requirements, and only the latter possess the signalling domain knowledge. To overcome this challenge, we adopted an iterative process of requirement elicitation. We started out by gathering background information and identifying hazards by interviewing signalling experts. These general hazards were translated to requirements in natural language. We then attempted to verify the refined requirements using the mCRL2 model checker. For requirements that did not hold for the model we assessed whether the model contains an error or the requirement is too strong. In the latter case we refined the requirement.

For example, we derived the following requirement: *"When component F_EST_EfeS_SR signals to component F_SCI_EfeS_Sec_SR that the object controller is not ready for a connection by sending a message on port 'T18_Not_Ready_For_PDI_Connection', then component F_SCI_EfeS_Sec_SR is not allowed to be in the state 'PDI_CONNECTION_ESTABLISHED' until a message on port 'T21_Ready_For_PDI_Connection' is received."* By formalising the requirement in the μ-calculus and checking whether the mCRL2 model satisfied this requirement, we found a counterexample in which the communication over port 'T18_Not_Ready_For_PDI_Connection' happens while F_SCI_EfeS_Sec_SR is in the state 'PDI_CONNECTION_ESTABLISHED'. Clearly, the component needs some time to process the message and move out of the state 'PDI_CONNECTION_ESTABLISHED'.

We weakened the requirement to *"When component F_EST_EfeS_SR signals to component F_SCI_EfeS_Sec_SR that the object controller is not ready for a connection by sending a message on port 'T18_Not_Ready_For_PDI_Connection', then component F_SCI_EfeS_Sec_SR will always eventually move out of the state 'PDI_CONNECTION_ESTABLISHED' and not establish a connection again until* *a message on port 'T21_ Ready_ For_ PDI_ Connection' is received.".*

When we are confident that the formula captures the intended purpose, but it still does not hold for the model we go back to the signalling engineer to discuss the results. An important tool in understanding why a requirement does not hold is the counterexample functionality in mCRL2 [30, 116], which provides the fragment of the LTS (dis)proving the formula. Sometimes the counterexample is not very useful because it contains a very large part of the LTS but other times it reflects a concrete scenario that can also be easily communicated to the signalling engineer.

For the SCI-LX interface the requirements have been derived by us, by inspecting the SysML model. For the other interfaces we formalised requirements in collaboration with railway experts from ProRail, and for the generic interface also with experts from DB.

5.2.2 Adding Selfloops

The semantics of mCRL2 is action-based and thus abstracts from the contents of a state; states are only distinguished by their transitions. The logic, which in the case of the mCRL2 toolset is the modal μ-calculus, also only refers to transitions. For models that are derived from state-based formalisms, such as SysML, it can be desirable to also consider the contents of a state. A way to achieve this is by adding selfloops to states, which exposes information in the states in the labels of transitions. We added the following three types of selfloops.

1. A selfloop showing which SysML states components are in: `inState(`c`, `s`)`, where c is the name of a component and s the name of a state machine state.

2. A selfloop showing which events are in the event queue: `inEventPool(`c`, `e`)`, where c is the name of a component and e an event.

3. A selfloop showing that some event queue is full: `eventPoolFull`, which has no parameters. This allows us to check for deadlocks due to two components with a full event queue trying to send a message to each other. Such deadlocked states can be identified by the formula

   $[\top^*]$`<eventPoolFull>`$\top$.

4. A selfloop showing the valuation of variables: `varVal(`c`, `n`, `v`)`, where c is the name of a component, n the name of the variable and v the value assigned to it.

States that were behaviourally equivalent may become distinct by the selfloops. Hence, by adding the selfloops the minimised state space becomes bigger.

A more in depth study on LTSs in which the states contain information in the form of global variables, is performed in Chapter 7.

5.2.3 Common Verification Approach

For each interface we use a similar verification approach. We discuss the general approach in this section. In each case study we present additional measures to reduce the state space.

For each interface we enable pulse packs and the resetting of data parameter ports to their initial value once a pulse pack has been handled (as discussed in Section 4.1.2). We also split the generic and specific parts of each interface, with the exception of the SCI-LX interface. We use the compositional minimisation approach of Chapter 4 to generate the LTS.

By reducing the state space modulo branching bisimulation we remove divergence (i.e. infinite internal activity) if there is any. This is a concern for the preservation of properties, especially for liveness requirements. In our toolchain we hide all the actions related to SysML transition selection and execution, but do not hide communication between components. For the SysML models considered in this chapter, any loop of behaviour involves visible communication. Hence our models do not contain divergence in the first place so minimising modulo branching bisimulation does not remove divergence. This might not be the case for future applications of our toolchain.

A number of bash scripts automate running the verifications. The scripts orchestrate the mCRL2 toolset. The mCRL2 model is first linearised, then the LTS is computed with compositional minimisation and, finally, for each μ-calculus formula in a designated folder a PBES is constructed and subsequently solved. The LTSs of components are explored in parallel and requirements are also verified in parallel. These scripts (and instructions on how to run them), the mCRL2 models and the μ-calculus formulas can be found in the Zenodo repository [12].

The following invocations of mCRL2 tools are used to generate the state space, assuming a SysML model with three parallel state machines.

```
mcrl22lps model.mcrl2 model.lps
lpssuminst model.lps model.2.lps -f -rjittyc
lpssumelm model.2.lps model.3.lps
lpsrewr model.3.lps model.linearized.lps
lpscleave model.lps comp0.lps rest.lps -m -a -c -t -fc0
  -psc_StateMachine,eq_StateMachine,mon_StateMachine,
    vars_StateMachine,ms_StateMachine,behav_StateMachine,
    exc_StateMachine,mq_StateMachine,head_mq_StateMachine,
    reset_StateMachine
lpscleave rest.lps comp1.lps comp2.lps -m -a -c -t -fc1
  -psc_StateMachine1,eq_StateMachine1,mon_StateMachine1,
    vars_StateMachine1,ms_StateMachine1,behav_StateMachine1,
    exc_StateMachine1,mq_StateMachine1,head_mq_StateMachine1,
    reset_StateMachine1
lps2lts --cached -rjittyc comp0.lps comp0.lts
lps2lts --cached -rjittyc comp1.lps comp1.lts
lps2lts --cached -rjittyc comp2.lps comp2.lts
```

```
ltsconvert comp0.lts comp0.min.lts -ebranching-bisim
ltsconvert comp1.lts comp1.min.lts -ebranching-bisim
ltsconvert comp2.lts comp2.min.lts -ebranching-bisim
lpscombine comp1.min.lts comp2.min.lts combined12.lts
  --introduce-tags -l -fc1
lpscombine comp0.min.lts combined12.lts statespace.lts
  --introduce-tags -l -fc0
ltsconvert statespace.lts statespace.min.lts -ebranching-bisim
```

To verify a formula with or without counterexample generation we use the following sequence of commands respectively.

```
lts2pbes statespace.min.lts -p -c --formula=req.mcf req.pbes
pbessolve req.pbes --in=pbes --file=statespace.min.lts
  -rjittyc -s2

lts2pbes statespace.min.lts -p --formula=req.mcf req.pbes
pbessolve req.pbes --in=pbes -rjittyc -s2
```

Running the verifications does not require many CPU cores but does require a significant amount of RAM. We have observed memory usage in excess of 2TB.

5.2.4 Sanity Checks

Potentially the mCRL2 model is flawed due to mistakes made while entering the SysML diagrams in the translation framework or due to flaws in the automated translation. Verifying “sanity check” requirements is one way to test whether the mCRL2 model is as intended. An easy requirement is that all SysML states are reachable. This requirement can detect, and has detected, many flaws such as missing transitions, wrong evaluations of guards, etcetera that have the effect that certain SysML states are unreachable. In Section 5.7 we discuss more ways to validate that the mCRL2 model is correct. Below we provide the reachability requirement for the SCI-LX level crossing interface. To check it `inState` selfloops need to be added for every state.

ID	SANITY_LX_1
Summary	Every SysML state is reachable
Detailed description	Sanity check whether all SysML states are reachable in the mCRL2 model. Some errors in the jEULYNX encoding or translation can be caught in this manner. Pseudostates are excluded as they are often skipped over when taking a compound transition.

μ-calculus formula

```
∀ c:CompName,s:StateName. val(
  c ∈ [BEQ_prim,BEQ_prim_LX,BEQ_scpPrim,BEQ_scpSeec,
      BEQ_seec,BEQ_seec_LX]
  ∧ s ∈ states(SMDefs(c))
  ∧ (is_simple(stateInfo(SMDefs(c))(s))
    ∨ is_composite(stateInfo(SMDefs(c))(s)))
) => <⊤*.inState(c,s)>⊤
```

Initially, this sanity check formula did not hold for the SCI-LX model. The tool mCRL2 produces a counterexample showing which part of the state space invalidates the formula. Unfortunately, the counterexample was not very helpful as it simply contained the entire state space. As mentioned the `inState` selfloops are enabled for all state names. Using `ltsinfo -a` we were able to determine which `inState` labels were unreachable. This revealed that the states PDI_VERSION_-UNEQUAL and PDI_CHECKSUM_UNEQUAL were not reachable in one of the components. The version and checksum are both parameters set by the environment for components at the interlocking and field element side. Our restrictions on the environment only allowed them to have the same version and checksum information. We relaxed the restrictions and allowed the level crossing to switch between two version and checksum values, allowing both the flow where they are equal and where they differ. With this change the formula holds. This shows that such general ‘sanity’ requirements are a way to find model/configuration errors.

Note that for the final model we did not enable all `inState` selfloops. Hence, the formula does not hold for the model as it is in the Zenodo repository [12].

5.3 Adjacent Level Crossing Interface (SCI-LX)

The SCI-LX interface is the odd one out in our analysis. Unlike the other interfaces SCI-LX is not a subsystem interface but an *adjacent* system interface. The generic layer of EULYNX for adjacent systems is less complex. The SCI-LC interface is the subsystem variant for the level crossing. DB Netz uses SCI-LX, Prorail uses SCI-LC.

The functionality of the level crossing interface is comparable to other interfaces. The interlocking can send commands to the level crossing and the level crossing reports its status. The commands are related to (de)activation. When the level crossing is activated it follows a sequence of ringing a bell, flashing lights and closing the barriers. What this activation looks like exactly differs per country and location and is outside the scope of EULYNX. Status reports from the level crossing report on the current activation status, possible failures and obstacle detection information for level crossings that support it.

The SCI-LX interface consists of 8 blocks in total, which includes the generic layer. Figure 5.4 gives an overview of the blocks and their relations and shows several layers in the specification. Blocks ‘scpSec’ and ‘scpPrim’ together model the RaSTA connection. Blocks ‘sec’ and ‘prim’ are generic blocks defining the connection management between adjacent EULYNX systems; they interface with the RaSTA protocol and handle errors such as mismatched protocol versions, telegram errors, etcetera. The bottom four blocks model the specifics of the LX interface. The blocks ‘S_SCI_LX_Sec_National_SR’ and ‘S_SCI_LX_-Prim_National_SR’ are not part of the EULYNX standard as they are country specific.

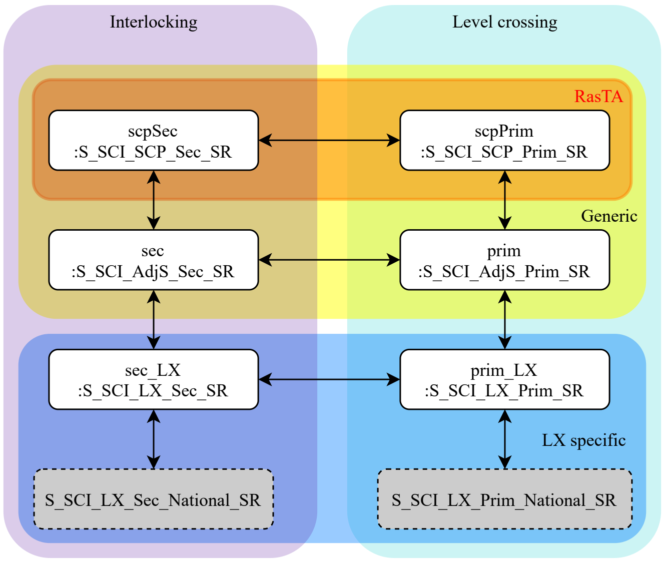


Figure 5.4: An overview of the blocks in the SCI-LX interface. Arrows indicate the existence of port connections between those blocks.

As ProRail does not implement SCI-LX, and DB has not yet made their implementation of these country specific blocks available to us we consider the country specific blocks to be part of the environment. The main functionality of ‘sec_LX’ and ‘prim_LX’ is forwarding commands from the interlocking to the level crossing and status reports from the level crossing to the interlocking between ‘S_SCI_LX_Sec_National_SR’ and ‘S_SCI_LX_Prim_National_SR’.

Due to the omission of these country specific blocks, and the fact that the generic adjacent interface contains fewer components, we can combine both the generic and interface specific layers in one model and still perform verification.

5.3.1 Requirements

We have derived three requirements for the LX interface. In general we have tried to distil the intended behaviour of the system as a whole by looking at the state machines of individual components. For example, the SysML model clearly specifies that when certain errors occur, the connection is aborted and the other side is notified. We derived that the intended behaviour is that when an error occurs, both the level crossing and the interlocking close the connection before reattempting a new connection, i.e. it should not be the case that either the object controller or interlocking is in a state reflecting that the connection is still up while it is actually not the case.

(No) Deadlock. The first requirement for the LX interface is that a connection always remains possible in the future. It should not be the case that the protocol gets stuck and can never reach a state again where a connection is established.

ID	REQ_LX_1
Summary	A connection always remains possible.
Detailed description	It always remains possible to reach a state where the components 'prim', 'sec', 'prim_LX' and 'sec_LX' are in their state PDI_CONNECTION_ESTABLISHED.

μ-calculus formula

```
[⊤*]([⊤*]<eventPoolFull>⊤ ∨ <⊤*>(
  <inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>⊤
  ∧ <inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>⊤
  ∧ <inState(BEQ_prim_LX,PDI_CONNECTION_ESTABLISHED)>⊤
  ∧ <inState(BEQ_seec_LX,PDI_CONNECTION_ESTABLISHED)>⊤))
```

Close Connection After Errors. When either the interlocking or the level crossing enters an error state, then the connection should be closed, and subsequently both need to enter their state PDI_CONNECTION_CLOSED before they reattempt a connection.

Both the interlocking and level crossing have a 'point of no return' after which they can move to PDI_CONNECTION_ESTABLISHED without interaction from the other side. For the interlocking this is after receiving the message T25_Msg_-Status_Report_Completed. For the level crossing this is after receiving message T24_Msg_Status_Report_Completed. We needed to adjust the mCRL2 model by adding `inEventPool` selfloops to be able to express these properties more easily.

ID	REQ_LX_2
Summary	Being in an error state implies that the connection is closed.
Detailed description	When component prim or sec is in a state PDI_INIT_-TIMEOUT, PDI_VERSION_UNEQUAL, PDI_CHECK-SUM_UNEQUAL, PDI_PROTOCOL_ERROR or PDI_-TELEGRAM_ERROR then the components 'prim' and 'sec' must both enter their state PDI_CONNECTION_-CLOSED before they can reach the state PDI_CONNEC-TION_ESTABLISHED.

μ-calculus formula

```
[⊤*](((∃ s:StateName. (val(s ∈ [PDI_INIT_TIMEOUT, PDI_VERSION_UNEQUAL,
    PDI_CHECKSUM_UNEQUAL, PDI_PROTOCOL_ERROR, PDI_TELEGRAM_ERROR]))
    ∧ (<inState(BEQ_prim,s)>⊤ ∨ <inState(BEQ_seec,s)>⊤))
  ∧ (∀ s:StateName. val(s ∈ [CHECKING_PRIM_STATUS,
    PDI_CONNECTION_ESTABLISHED]) => [inState(BEQ_seec,s)]⊥)
  ∧ [inEventPool(BEQ_seec,ChangeEventWithDataParams(
    [ASALA_PushGlobalVar(T25_Msg_Status_Report_Completed)],[]))]⊥
  ∧ (∀ s:StateName. val(s ∈ [CHECKING_SEC_STATUS,SENDING_PRIM_STATUS,
    WAITING_FOR_INIT_COMPLETION,PDI_CONNECTION_ESTABLISHED])
    => [inState(BEQ_prim,s)]⊥)
  ∧ [inEventPool(BEQ_prim,ChangeEventWithDataParams(
    [ASALA_PushGlobalVar(T24_Msg_Status_Report_Completed)],[]))]⊥)
    => ∀ c:CompName. (val(c ∈ [BEQ_prim, BEQ_seec]))
    => (νX(closed: Bool = ⊥). ((<inState(c,PDI_CONNECTION_CLOSED)>⊤
        ∧ [⊤]X(⊤))
      ∨ ([inState(c,PDI_CONNECTION_CLOSED)]⊥ ∧ [⊤]X(closed)))
  ∧ ((val(closed)) ∨ [inState(c,PDI_CONNECTION_ESTABLISHED)]⊥)))
```

Exchanging Status Updates and Commands. The main functionality of the LX interface (besides connection management) is exchanging messages. Below we formulate a requirement that these messages are actually delivered. Note that we abbreviate actions of the shape `send(c,p,v)|receive(c2,cp,v)` to `send(c,p,v)` in this thesis. The sending port is a unique identifier for the communication channel. We could therefore indeed omit the `receive` actions in the formulas if we would also hide the `receive` actions in the LTS.

ID	REQ_LX_3
Summary	Status updates and commands are delivered on the other side.

Detailed description	When the component 'prim_LX' receives a message T20_-Internal_Input whilst the connection is established it will result in an output T21_Internal_Output by 'sec_LX'. The message may only be dropped if the connection is terminated before the message is delivered. Similarly, when 'sec_LX' receives a message T20_Internal_Input whilst the connection is established it will result in an output T21_-Internal_Output by 'prim_LX'.

μ-calculus formula

```
%Messages from the level crossing are delivered to the interlocking
[⊤*](∀ v:Value. [send(CompPortPair(ENV_prim_LX, T20_Internal_Input),
    Value_Pulse_Pack([VarValuePair(DT20_Type,v)]))](
  (<inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>⊤
    ∧ <inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>⊤
    ∧ <inState(BEQ_prim_LX,PDI_CONNECTION_ESTABLISHED)>⊤
    ∧ <inState(BEQ_seec_LX,PDI_CONNECTION_ESTABLISHED)>⊤)
  => (νX.(
    %As long as the message has not been delivered recurse using X
    [(¬(∃ v2:Value. send(CompPortPair(BEQ_seec_LX, T21_Internal_Output),
        Value_Pulse_Pack([VarValuePair(DT21_Type,v2)]))))]X
    %There must exist a path ending in a delivered message
    ∧ (<(¬(∃ v2:Value.send(CompPortPair(BEQ_seec_LX,T21_Internal_Output),
        Value_Pulse_Pack([VarValuePair(DT21_Type,v2)]))))*
    .send(CompPortPair(BEQ_seec_LX, T21_Internal_Output),
        Value_Pulse_Pack([VarValuePair(DT21_Type,v)]))>⊤))
  %Or Prim_LX is destined to leave PDI_CONNECTION_ESTABLISHED
  ∨ ((<(¬(∃ c:CompName, p1,p2:VarName, v2:Value.
    send(CompPortPair(c,p2),v2)))*>
    ([inState(BEQ_prim_LX,PDI_CONNECTION_ESTABLISHED)]⊥))
  %Or Sec_LX is destined to leave PDI_CONNECTION_ESTABLISHED
  ∨ (<(¬(∃ c:CompName, p1,p2:VarName, v2:Value.
    send(CompPortPair(c,p2),v2)))*>
    ([inState(BEQ_seec_LX,PDI_CONNECTION_ESTABLISHED)]⊥))
  %Or there is a component which is deadlocked due to a full event pool
  ∨ ([⊤*]<eventPoolFull>⊤)))))

%Symmetric requirement for messages from interlocking to level crossing
...
```

5.3.2 Verification Approach

To restrict the state space whilst maximising the behaviour covered in our analysis we have settled on the following configuration of the translation.

Optimisation Settings. Ports open to the environment are made synchronous but there is no restriction on when the environment can send a message. `inEventPool` selfloops are enabled for two events, that are referenced by requirement REQ_-LX_2. In addition, as mentioned earlier, an action `eventPoolFull` without data

Requirement	Result	Time no counter	Time counterexample
REQ_LX_1	true	108m	
REQ_LX_2	false	126m	1,496m
REQ_LX_3	true	789m	

Table 5.9: Verification results for the SCI-LX interface. The last two columns respectively show the time in minutes to verify a requirement with and without the counterexample generation feature. In both cases the time needed by the tools `lts2pbes` and `pbessolve` is combined.

parameters is used to detect deadlocks. Eleven `inState` selfloops are enabled, as they are referenced by the requirements.

Restrictions Configuration Ports. The configuration ports are disabled as they cannot change value during execution. The ports to configure timeout values are set to a fixed value (the specific value is irrelevant as we abstract from time in the mCRL2 model). The ports to configure whether checksum data is used are set to true. Similarly to the generic subsystem model there are also ports to configure version and checksum data. For the 'sec' component these are set to fixed values, "V1" and "D1" respectively. To include all scenarios where the version and checksum are the same or differ, the 'prim' component can switch between version "V1" and "V2" and checksum data "D1" and "D2".

5.3.3 Results

Linearising, and compositionally constructing the state space took 91 minutes, resulting in an LTS with 16.8 million states. Verifying the requirements (in parallel) without counterexample generation took 13 hours. Table 5.9 shows the results for the final model. Some requirements did not hold initially, revealing flaws in the model.

Model checking requirement REQ_LX_2 revealed a mistake in the SysML model. The counterexample revealed an issue in the modelling of the RaSTA blocks. The state machines of the RaSTA blocks, which we obtained from signalling engineers, did not behave as specified by the RaSTA specification. We adjusted the state machines to closely reflect the behaviour of RaSTA, while still abstracting from details such as sequence numbers. As these state machines are not part of EULYNX this does not imply a mistake in EULYNX specifications.

Adjusting the state machines of RaSTA triggered us to inspect the interface between EULYNX and RaSTA more closely. This inspection resulted in the discovery of a discrepancy in the 'sec' side of both the generic adjacent specification and the generic subsystem specification, i.e. the issue is present in all EULYNX interfaces. The ‘sec’ blocks in the generic interfaces have three ports to interface with RaSTA:

- 2 input: SCP_Connection_Established and SCP_Connection_Terminated,
- 1 output: Terminate_SCP_Connection

They do not have an output port to signal that the connection should be established, which is present for the ‘prim’ side. However, the RaSTA specification specifies that the RaSTA protocol starts in the “Closed” state. An “Open Connection” request is needed to move to the state “Down”, from which a connection can be established. In order to make the interface modelled in EULYNX compatible with the interface that RaSTA offers, an extra port Establish_SCP_Connection is needed, in the same way the prim side has such a port.

Requirement REQ_LX_1 does not hold for the model: there exists a deadlock in the model, stemming from the combined behaviour of the SCP blocks and the generic adjacent blocks. Hence, it is not only relevant for the LX interface but for all EULYNX interfaces using the adjacent system approach. More specifically, the blocks S_SCI_SCP_Prim_SR and S_SCI_AdjS_Prim_SR can end up in deadlock. When the block S_SCI_AdjS_Prim_SR enters the state PDI_CONNECTION_CLOSED it sends a pulse on port T6_Establish_SCP_Connection to the SCP/RaSTA layer. The RaSTA layer sets up the connection and responds with T5_SCP_Connection_Established. If RaSTA fails to establish a connection, due to a timeout, S_SCI_AdjS_Prim_SR is never notified of this failure and cannot take a new initiative to establish a connection, hence we have reached a deadlock. Further investigation showed that the exact same construction is used in the subsystem variant of the generic EULYNX layer.

We notified engineers from ProRail and DB about our findings. Engineers from DB proposed to resolve the issues with RaSTA by adding the following note: “It is assumed that the SCP layer handles each connection error by itself after sending the trigger on T6_Establish_SCP_Connection. A retriggering of SCP connection is not the responsibility of the SCI layer”. By inspecting the RaSTA specification we learned this remedy is not in compliance with the RaSTA standard. A custom implementation of RaSTA would be needed.

The assumptions on RaSTA should also be made more precise as it still leaves ambiguities. Is the RaSTA layer also allowed to automatically reconnect when an error occurs after T5_SCP_Connection_Established has been sent? EULYNX specifications seem to suggest that a new message on port T6_Establish_SCP_Connection is needed.

Furthermore, EULYNX specifications should not only make a note on the interface between the ‘prim’ component and RaSTA but also between the ‘sec’ component and RaSTA. On that side of the interface a slightly different assumption needs to be made: the RaSTA implementation on the server side of the RaSTA protocol should never wait for an ‘open connection’ event to establish a connection.

The RaSTA specification contains an event-state matrix, specifying in detail how each event should be handled in each state. We recommend specifying the exact changes that need to be made to the event-state matrix of RaSTA to make a RaSTA implementation EULYNX compliant. Presumably, an extra state “RETRY” needs to be added for the client side of RaSTA (the ‘prim’ side of EULYNX).

For the server side of RaSTA (the ‘sec’ side of EULYNX) the following assumption could be made: any event-state combination that leads to a transition to the ‘closed’ state should instead go to the ‘down’ state. Additionally, any such transition to the ‘down’ state should also result in setting the sequence number (SN_T) to a random value, which is normally done after an ‘open connection’ event.

We have implemented the change to the RaSTA blocks as suggested by DB. Requirement REQ_LX_1 holds under these assumptions. REQ_LX_2 does not hold for the adjusted model. The counterexample shows that while component sec is still in the state PDI_PROTOCOL_ERROR the RaSTA block on the sec side can already re-establish a RaSTA connection because it can always establish a connection. To be more specific, the counterexample produced by the mCRL2 toolset shows the following sequence of events:

1. Blocks ‘prim’ and ‘sec’ are initialising a connection and the RaSTA connection is up.
2. A protocol error occurs on the ‘sec’ side.
3. ‘sec’ moves to PDI_PROTOCOL_ERROR.
4. ‘sec’ signals to RaSTA that the connection needs to be closed.
5. The RaSTA connection closes.
6. ‘prim’ is notified of the closed RaSTA connection.
7. ‘prim’ signals to RaSTA to re-establish a connection.
8. ‘sec’ is notified that the RaSTA connection is closed
9. A new RaSTA connection is established.
10. ‘prim’ is notified that the RaSTA connection is up and moves to ESTABLISHING_PDI_CONNECTION.
11. ‘sec’ moves to PDI_CONNECTION_CLOSED.
12. ‘sec’ is notified that the RaSTA connection is up.
13. ‘sec’ moves to ESTABLISHING_PDI_CONNECTION.

Requirement REQ_LX_2 is violated as the ‘prim’ component does not go to PDI_CONNECTION_CLOSED between ‘sec’ being in the state PDI_PROTOCOL_ERROR and ‘prim’ reaching the state PDI_CONNECTION_ESTABLISHED. They do both first close the connection before another attempt is made, so this is not a serious issue.

We propose an alternative remedy that does not require modifications to RaSTA. We propose to let S_SCI_SCP_Prim_SR send a pulse on T5_SCP_-Connection_Terminated when it moves from the state START to the state CLOSED. Additionally, add an internal transition to the state PDI_Connection_Closed of block S_SCI_AdjS_Prim_SR which resends a pulse on T6_Establish_SCP_-Connection when it receives the pulse T5_SCP_Connection_Terminated. When the connection fails the RaSTA block moves to a state "closed" and will not open the connection again until a new message Establish_SCP_Connection is received. We have implemented this remedy in SysML. All formulas hold for the adjusted model. The SCI-LX and generic models in Zenodo both use the remedy where RaSTA is adjusted with a RETRY state, as that most closely resembles the current modelling by EULYNX.

5.4 Generic Subsystem Interface

The main role of the generic interface is connection management. Initialising the connection consists of a number of steps. Firstly, a RaSTA connection is established. Secondly, the interlocking and object controller message each other to request a connection and exchange the version of the EULYNX protocol that they are using. Finally, the interface specific layer is prompted to send an initial status update. When the interface specific layer has finished the initial status report, it reports this to the generic layer. After these initialisation steps, the connection is fully established and the interface specific layer of the interlocking and object controller can freely exchange messages.

During initialisation or when the connection is established, the connection can be aborted by either side. When they encounter an error (which can occur non-deterministically in the model), such as a power failure or timeout, they move to an error state and notify the other side by closing the RaSTA connection. The connection can then be re-established using the normal procedure.

The division in components is depicted in Figure 5.5. The role of each component is described below.

- The components F_SCI_SCP_Sec_SR and S_SCI_SCP_Prim_SR manage the RaSTA connection on the side of the object controller and on the side of the interlocking, respectively.

- The components S_SCI_EfeS_Prim_SR and F_SCI_EfeS_Sec_SR initiate and manage the connection.

- The component F_EST_EfeS_SR defines interaction with generic electronics, which includes boot-up behaviour and failure modes.

- The component F_SMI_EfeS_SR is used for maintenance access.

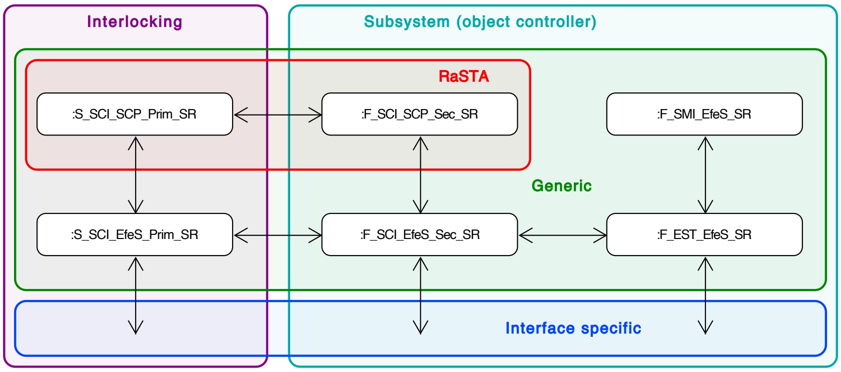


Figure 5.5: Structure of the components of the EULYNX generic subsystem interface. Arrows indicate communication channels between components.

5.4.1 Requirements

The main hazard related to the connection management is that a connection is wrongly established or maintained. The reason that this would pose a risk is that the interlocking would continue to think it has a connection with the field element while it is no longer updated on the status of the field element. A secondary hazard is that a connection is prevented from being established at all. This might not be a direct safety hazard but could disrupt train services.

We can derive the following hazards for the generic interface:

1. A connection is still established/maintained despite one of the following errors.

    - The object controller and interlocking are using a different version of the protocol;
    - A checksum sent during initialisation does not match;
    - A protocol error;
    - A telegram error;
    - A timeout during initialisation;
    - There is no operating voltage;
    - A hardware error;
    - The other side of the connection has closed the connection.

2. A deadlock or livelock prevents the system from establishing a connection.

These two hazards are addressed with multiple requirements, which are listed below. The second hazard is addressed by requirement REQ_PDI_4. The other requirements expand the first hazard by specifying different scenarios in which the object controller or interlocking is not allowed to establish a connection. Note that the numbering of requirements starts at 2. Requirement REQ_PDI_1 was scrapped because it was too trivial.

The identified hazards and requirements are based on discussions with signalling engineers and a list of requirements formulated by signalling engineers from DB.

ID	REQ_PDI_2
Summary	Do not establish a connection when the interlocking and object controller use a different protocol version.
Detailed description	Whilst establishing a connection, in the case that the object controller sees that the interlocking uses a different version of the protocol, it sends a message to the interlocking notifying the failed version check and moves to the state not ready for a connection.

μ-calculus formula

```
[⊤*.send(CompPortPair(BEQ_seec,T13_Msg_PDI_Version_Check),
    Value_Pulse_Pack([VarValuePair(DT13_Checksum_Data,
        Value_Custom(STR_NotApplicablee)),
      VarValuePair(DT13_Result,Value_Custom(STR_Not_Match))]))]
  %The object controller moves to the state Not_Ready_For_Connection
  %before it is allowed to succesfully establsish a connection
  ((νX. ([inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]⊥ ∧ [⊤]X)
    ∨ <inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>⊤)
  %The interlocking moves to the state Connection_closed
  %before it is allowed to succesfully establsish a connection
  ∧ (νX. ([inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)]⊥ ∧ [⊤]X)
    ∨ <inState(BEQ_prim,PDI_CONNECTION_CLOSED)>⊤))
```

ID	REQ_PDI_3
Summary	Disconnect when a checksum fails during initialisation.
Detailed description	Whilst establishing a connection, in the case that the interlocking receives an incorrect checksum (indicating a malformed message) from the object controller in the Msg_PDI_Version_Check message, the interlocking terminates the connection.

μ-calculus formula

```
[⊤*.inState(BEQ_prim,PDI_CHECKSUM_UNEQUAL)]
  %The object controller moves to the state Not_Ready_For_Connection
  %before it is allowed to succesfully establsish a connection
  ((νX. ([inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]⊥ ∧ [⊤]X)
    ∨ <inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>⊤ )
  %The interlocking moves to the state Connection_closed
  %before it is allowed to succesfully establsish a connection
  ∧ (νX. ([inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)]⊥ ∧ [⊤]X)
    ∨ <inState(BEQ_prim,PDI_CONNECTION_CLOSED)>⊤))
```

ID	REQ_PDI_4
Summary	A connection always remains possible.
Detailed description	As long as there is no telegram error (the reception of a malformed message, which may occur non-deterministically in the model) it always remains possible in the future to reach the state PDI_Connection_Established at both the interlocking and object controller side.

μ-calculus formula

```
[⊤*]( %in any state
  %we either have that PDI_Connection_Impermissible is inevitable
  (<inState(BEQ_prim,PDI_CONNECTION_IMPERMISSIBLE)>⊤
    ∨ <inState(BEQ_seec,PDI_CONNECTION_IMPERMISSIBLE)>⊤
    ∨ <inState(BEQ_prim,PDI_TELEGRAM_ERROR)>⊤
    ∨ <inState(BEQ_seec,PDI_TELEGRAM_ERROR)>⊤)
  %or we can eventually establish a connection
  ∨ (<⊤*>(
    <inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>⊤
    ∧ <inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>⊤)))
```

ID	REQ_PDI_5
Summary	Close the connection when a protocol or telegram error occurs.
Detailed description	At the moment a protocol error or telegram error occurs at either the side of the object controller or the interlocking, both the interlocking and the object controller will eventually move to PDI_Connection_Closed before reattempting a connection.

μ-calculus formula

```
[⊤*]((
  (<inState(BEQ_seec,PDI_TELEGRAM_ERROR)>⊤
    ∨ <inState(BEQ_seec,PDI_PROTOCOL_ERROR)>⊤)
  => (νX. ([inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]⊥ ∧ [⊤]X)
    ∨ <inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>⊤)
  ) ∧ ((<inState(BEQ_prim,PDI_TELEGRAM_ERROR)>⊤
    ∨ <inState(BEQ_prim,PDI_PROTOCOL_ERROR)>⊤)
  => (νX. ([inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)]⊥ ∧ [⊤]X)
    ∨ <inState(BEQ_prim,PDI_CONNECTION_CLOSED)>⊤)))
```

Requirement REQ_PDI_6 below has a slightly weaker variant called REQ_-PDI_6_1. This will be further explained in Section 5.4.3.

ID	REQ_PDI_6
Summary	Close connection when the object controller is unfit to operate.
Detailed description	When F_EST_EfeS_SR, which monitors the hardware of the object controller, signals it is not ready for a connection, the object controller will move to not ready for connection and only reattempts a connection after receiving a message ready for connection.

μ-calculus formula

```
[⊤*.send(CompPortPair(BEQ_eest,T18_Not_Ready_For_PDI_Connection),
    Value_Pulse_Pack([]))](
  %when in PDI_connection established, move to not ready for connection
  ((<inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>⊤)
    => (μX. (([¬(∃ c:CompName,s:StateName. inState(c,s))]X)
      ∨ [inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]⊥)))
  %do not establish a connection until allowed again by F_EST_EfeS_SR
  ∧ (νX. [¬send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),
      Value_Pulse_Pack([]))]X
    ∧ (<inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>⊤
    => ([(¬send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),
        Value_Pulse_Pack([])))*]
    [inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]⊥))))
```

Variant ID	REQ_PDI_6_1
Variant summary	Do not establish a new connection when the object controller is unfit to operate.
Detailed description variant	When F_EST_EfeS_SR, which monitors the hardware of the object controller, signals it is not ready for a connection, and the object controller has moved out of the state PDI_-CONNECTION_ESTABLISHED no new connection will be established until a new message from F_EST_EfeS_SR indicates that the object controller is ready again.

μ-calculus formula

```
[⊤*.send(CompPortPair(BEQ_eest,T18_Not_Ready_For_PDI_Connection),
    Value_Pulse_Pack([]))]
  (νX. [¬send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),
      Value_Pulse_Pack([]))]X
  ∧ (<inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>⊤
  => ([(¬send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),
      Value_Pulse_Pack([])))*]
    [inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]⊥)))
```

ID	REQ_PDI_7
Summary	Abort connection attempt when a timeout occurs during initialisation.
Detailed description	When S_SCI_EfeS_Prim_SR does not reach the state PDI_Connection_Established after entering state Establishing_PDI_Connection within the time D2_Con_tmax_-PDI_Connection, both S_SCI_EfeS_Prim_SR and F_-SCI_EfeS_Sec_SR reach PDI_Connection_Closed before reattempting a connection.

μ-calculus formula

```
[⊤*](<inState(BEQ_prim,PDI_INIT_TIMEOUT)>⊤ =>
  %The interlocking moves to the state Connection_closed
  %before it is allowed to succesfully establsish a connection
  (νX. ([inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)]⊥ ∧ [⊤]X)
    ∨ <inState(BEQ_prim,PDI_CONNECTION_CLOSED)>⊤))
```

ID	REQ_PDI_8
Summary	When the object controller or interlocking closes the connection then so does the other side.
Detailed description	When S_SCI_EfeS_Prim_SR and F_SCI_EfeS_Sec_SR are both in state PDI_Connection_Established, and one leaves that state, the other will eventually leave the state as well.

μ-calculus formula

```
[⊤*]((
  <inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>⊤
  ∧ <inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>⊤
) =>[⊤](((¬<inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>⊤)
  => μX. [¬(∃ c:CompName,s:StateName. inState(c,s))]X
    ∨ (<inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>⊤))
  ∧ ((¬<inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>⊤)
    => μX. [¬(∃ c:CompName,s:StateName. inState(c,s))]X
      ∨ (<inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>⊤))))
```

5.4.2 Verification Approach

Recall the options to restrict the state space discussed in Section 4.1. To restrict the state space whilst maximising the behaviour covered by our analysis we have settled on the following configuration of the translation.

Optimisation Settings. Ports open to the environment are made synchronous but there is no restriction on when the environment can send a message, see Section 4.1.3. `inEventPool` selfloops are disabled as they are not referenced by the requirements. All `inState` selfloops are enabled as many state names are referenced by the requirements.

Restrictions Configuration Ports. The configuration ports are disabled as they cannot change value during execution. There are ports that configure timeout values are set to a fixed value (the specific value is irrelevant as we abstract from time in the mCRL2 model). The ports to configure whether checksum data is used are set to true. Also the port D20_Con_MDM_Used, which enables interaction with the SMI component, is set to true. There are also ports to configure version and checksum data. For the F_SCI_EfeS_Sec_SR component these are set to fixed values, “V1” and “D1” respectively. To include all scenarios where the version and checksum are the same or differ, the S_SCI_EfeS_Prim_SR component can switch between version “V1” and “V2” and checksum data “D1” and “D2”.

Other Restrictions Environment Ports. We have disabled the port T22_-Content_Telegram_Error for both the S_SCI_EfeS_Prim_SR and F_SCI_EfeS-

Requirement	**Result**	**Time no counter**	**Time counterexample**
REQ_PDI_2	true	80m	
REQ_PDI_3	true	89m	
REQ_PDI_4	false	47m	713m
REQ_PDI_5	true	87m	
REQ_PDI_6	false	118m	998m
REQ_PDI_6_1	true	87m	
REQ_PDI_7	true	49m	
REQ_PDI_8	true	135m	

Table 5.20: Verification results for the generic interface. The last two columns respectively show the time in minutes to verify a requirement with and without the counterexample generation feature. In both cases the time needed by the tools `lts2pbes` and `pbessolve` is combined.

_Sec_SR component. The behaviour upon receiving a message on this port is identical to the behaviour upon receiving a message on T21_Formal_Telegram_-Error, this port is enabled).

5.4.3 Results

Linearising, and compositionally constructing the state space took 50 minutes, resulting in an LTS with 12.6 million states. Table 5.20 shows the results of verification and the running time for each requirement.

Requirement REQ_PDI_4, stating that a connection always remains possible, does not hold. The counterexample shows that two components can deadlock when trying to send a message to each other. Sending a message usually results in adding an event to the event queue (see [17]). The event queue mostly acts as a communication buffer, though state machines may also add events to their own queue by triggering change events. The event queue of state machines is finite in our models; when the event queue of a state machine is full, communication is no longer possible. When two components with a full event queue want to send a message to each other, they get into a deadlock. In EULYNX it is assumed that event queues are unbounded. However, unbounded event queues would cause an infinite state space due to communications from the environment and therefore make model checking unfeasible. Moreover, an event queue of arbitrary length can always be filled due to communications from the environment. Hence, increasing the size of the event queue will not remove the deadlock. We conjecture that in a model that includes timing, the probability of a deadlock is inversely proportional to the size of the event queue. However, it would be even better if the system is designed to be more robust against bursts of communication. We have made the recommendation to EULYNX to explicitly specify what happens when a buffers becomes full. Note that the formula for requirement REQ_LX_1, which was formulated at a later point in time than REQ_PDI_4, leaves out runs of the system ending up in a deadlock due to full event queues.

Requirement REQ_PDI_6 – stating that when the object controller is not ready for a connection, the connection is closed and not established again until it is ready – also does not hold. The counterexample produced by the explicit state tools shows us that a component may not always eventually move to PDI_-CONNECTION_CLOSED due to a loop of behaviour of another component. The other component loops by receiving a value from the environment over and over again. The requirement might hold under a mild component-based progress assumption, such as *justness* [18, 49], which excludes unrealistic computations in which a component never gets the chance to make progress. Requirement REQ_-PDI_6_1, which just checks whether a new connection is not established, does hold. We will revisit this requirement in Chapter 6, where we will embed a justness assumption in the formula.

5.5 Point Interface (SCI-P)

A railway point, also known as a 'turnout' or 'switch', is a mechanical safety-critical installation enabling trains to be guided from one set of rails to another (see Figure 5.6). Although implementations of points are country-specific, they all consist of one or more movable elements, which are controlled by point machines and monitored by sensors. Point machines are essentially engines moving the movable parts of a point. The positions of the movable elements determine the position of the point itself, namely 'left' or 'right' (see Figure 5.7), or 'neither' when changing from one to the other.

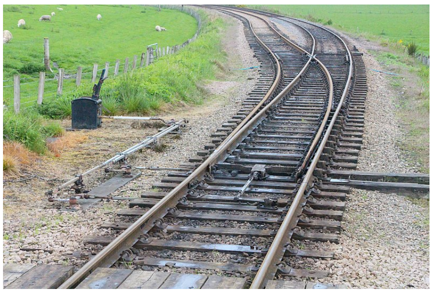


Figure 5.6: Point at Broomhill station, Scotland. Licensed under the Creative Commons Attribution-Share Alike 4.0 International license [117].

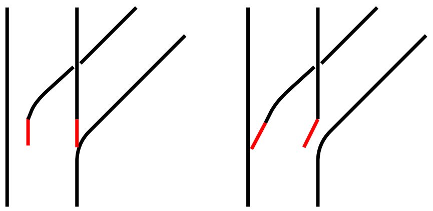


Figure 5.7: Schematic view of point positions, with movable elements in red. The left picture shows a point in the 'left' position, the right picture shows a point in the 'right' position.

In traditional points, the motors and sensors of the point machines are directly connected to the interlocking. With EULYNX a local object controller steers the point machines. The object controller provides a EULYNX compliant interface to the interlocking.

The two functions of the point-specific EULYNX interface are steering the point to a requested position and reporting the current position of the point to the interlocking. The output from the object controller to the point machines is either 'left', 'right' or 'stop'. The point machines, in turn, send back the current position of the point. The position can have three possible values, 'end position right', 'end position left' or 'no end position'. Since each point machine detects and reports its own position, there may not be consensus between point machines on the point position, in which case the object controller reports a 'no end position' message to the interlocking.

Division in Components. The EULYNX point specific interface is modelled using three components, see Figure 5.8. A pair of interlocking/object controller components (S_SCI_P_SR and F_SCI_P_SR) implement the exchange of point-specific commands and messages. Component F_SCI_P_SR relays point-specific messages to F_P3_Gen and ensures that the position is reported in the initialisation phase. The remaining component F_P3_Gen interacts with the point machines (which are outside the scope of EULYNX). The component F_P3_Gen performs two tasks in parallel: it monitors the current positions of the point machines and reports the information to F_SCI_P_SR, and it steers the point to the position that was most recently received from F_SCI_P_SR. When the generic layer reports a loss of connection or another failure, F_P3_Gen stops reporting the position or moving the point.

5.5.1 Requirements

The two main hazards related to points are derailments and train-train collisions. The hazard of derailment has many aspects. Physical failures of the track might

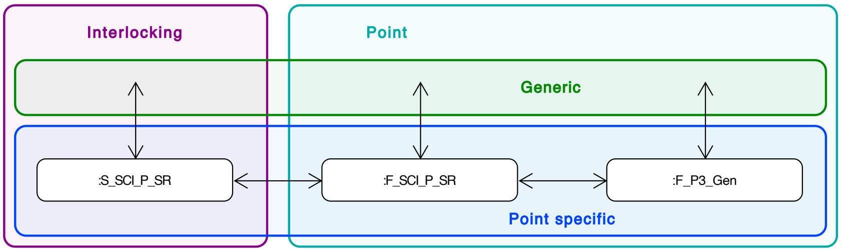


Figure 5.8: Structure of the components of the EULYNX point specific interface. Arrows indicate communication channels between components.

lead to derailment. A high speed in a tight curve could lead to derailment. Another possible cause is when a train goes over a point that is not in a proper (left or right) end position. Since the object controller of the point controls the movement of the point and informs the interlocking on the current position, correct behaviour of the point object controller is essential to prevent derailments.

Since points determine the routes of trains, correct behaviour of the object controller is also essential in preventing train-train collisions. The considerations for the object controller are again the correct control of the movement and reporting of the position. The principle for movement of the point is that only the interlocking knows when it is safe to move a point, so only the interlocking can initiate a movement. Two principles apply to reporting the position of the point: the interlocking should always be kept up to date concerning the position of a point; and when the position of a point is unknown, it is always assumed not to have an end position (meaning that it is unsafe to drive over the point).

We can derive the following hazards for the point specific interface:

1. The object controller reports an end position that is not accurate.
2. The object controller initiates a movement that was not expected by the interlocking.

These two hazards are addressed by the following requirements and are based on interviews with signalling engineers. Note that REQ_P_1 has a variant called REQ_P_1_1 which will be discussed in Section 5.5.3.

ID	REQ_P_1
Summary	The object controller must report changes in position.
Detailed description	The position of the point is determined by combining the input from the point machines. When all point machines report right the position is right. When all point machines report left the position is left. In any other case the point does not have an end position. When the position of the point is different from the last reported position, the new position must be reported to the interlocking. This obligation is lifted if communication with the interlocking is not possible due to a connection problem, power failure, etc. If the object controller cannot report the changed position, the connection with the interlocking must be closed, so that the interlocking knows that the position is unknown.

μ-calculus formula

```
[⊤ *]((%when the point is in an end position
  <inState(BEQ_pe51 , ALL_LEFT)>⊤ ∨ <inState(BEQ_pe51 , ALL_RIGHT)>⊤
) => % When point machine reports end position 'none ':
  ([send(CompPortPair(ENV_pe51, D21_PM1_Position),
      Value_Custom(STR_NO_END_POSITION))]
    (μX. [¬( ∃ st: StateName , c: CompName . inState (c,st))]X
      ∨ <send(CompPortPair(BEQ_sp , T20_Point_Position),
          Value_Pulse_Pack([VarValuePair(DT20_Point_Position,
              Value_Custom(STR_NO_END_POSITION))]))>⊤
    ∨ <inState( BEQ_pe51, WAITING_FOR_INITIALISING)>⊤
    ∨ [ inState(BEQ_fp, PDI_CONNECTION_ESTABLISHED)] ⊥
    ∨ [ inState(BEQ_sp, PDI_CONNECTION_ESTABLISHED)] ⊥)))
```

Variant ID	REQ_P_1_1
Variant summary	The object controller must report changes in position when there are no messages coming from the environment.
Detailed description variant	The same as the original requirement with the difference that the status update must always eventually happen when we only consider paths where the environment does not send inputs to the system.

μ-calculus formula

```
[⊤*](( %when the point is an end postion
  <inState(BEQ_pe51, ALL_LEFT)>⊤ ∨ <inState(BEQ_pe51, ALL_RIGHT)>⊤
) => % When point machine reports end position 'none':
  ([send(CompPortPair(ENV_pe51, D21_PM1_Position),
      Value_Custom(STR_NO_END_POSITION))]
    % We examine all paths for which the environment is silent
    (μX. [¬(∃ st:StateName, c, env: CompName, p1, p2: VarName, v: Value.
     (val(env ∈ [ENV_sp, ENV_fp, ENV_pe51])
       ∧ send(CompPortPair(env, p2), v))
     ∨ inState(c,st) ∨ eventPoolFull)]X
     %eventually no_end_position is reported to the interlocking
     ∨ <send(CompPortPair(BEQ_sp, T20_Point_Position),
         Value_Pulse_Pack([VarValuePair(DT20_Point_Position,
             Value_Custom(STR_NO_END_POSITION))]))>⊤
    %or the connection is no longer up
    ∨ <inState(BEQ_pe51,WAITING_FOR_INITIALISING)>⊤
    ∨ [inState(BEQ_fp,PDI_CONNECTION_ESTABLISHED)]⊥
    ∨ [inState(BEQ_sp,PDI_CONNECTION_ESTABLISHED)]⊥
    ∨ [⊤*]<eventPoolFull>⊤)))
```

ID	REQ_P_2
Summary	The object controller must not change position unless commanded by the interlocking.
Detailed description	The object controller may only instruct the point machines to move when a move is commanded by the interlocking. A movement command from the interlocking gives an authorisation to initiate movement in a certain position that ends when either a timeout occurs, the end position is reached or a movement command for the opposing position is sent.

μ-calculus formula

```
νX(allowed: Bool = ⊥, direction: Value = Value_Custom(STR_LEFT)).
  [send(CompPortPair(BEQ_sp,T1_Cd_Move_Point),
       Value_Pulse_Pack([VarValuePair(DT1_Move_Point_Target,
           Value_Custom(STR_LEFT))]))]
     X(⊤,Value_Custom(STR_LEFT))
  ∧[send(CompPortPair(BEQ_sp,T1_Cd_Move_Point),
      Value_Pulse_Pack([VarValuePair(DT1_Move_Point_Target,
          Value_Custom(STR_RIGHT))]))]
    X(⊤,Value_Custom(STR_RIGHT))
  ∧ [send(CompPortPair(BEQ_fp,T3_Msg_Timeout),
      Value_Pulse_Pack([]))]X(⊥,direction)
  ∧ [send(CompPortPair(BEQ_sp,T2_Msg_Point_Position),
      Value_Pulse_Pack([VarValuePair(DT2_Point_Position,direction)]))]
    X(⊥,direction)
  ∧ ((val(¬allowed ∨ (direction ≠ Value_Custom(STR_RIGHT)))) =>
    [send(CompPortPair(BEQ_pe51,D11_Move_Right),Value_Bool(⊤))]⊥)
  ∧ ((val(¬allowed ∨ (direction ≠ Value_Custom(STR_LEFT)))) =>
    [send(CompPortPair(BEQ_pe51,D10_Move_Left),Value_Bool(⊤))]⊥)
  ∧ [¬(send(CompPortPair(BEQ_sp,T1_Cd_Move_Point),
      Value_Pulse_Pack([VarValuePair(DT1_Move_Point_Target,
          Value_Custom(STR_LEFT))])))
    ∧ ¬(send(CompPortPair(BEQ_sp,T1_Cd_Move_Point),
        Value_Pulse_Pack([VarValuePair(DT1_Move_Point_Target,
            Value_Custom(STR_RIGHT))])))
    ∧ ¬(send(CompPortPair(BEQ_fp,T3_Msg_Timeout),Value_Pulse_Pack([])))
    ∧ ¬(send(CompPortPair(BEQ_sp,T2_Msg_Point_Position),
      Value_Pulse_Pack([VarValuePair(DT2_Point_Position,direction)])))]
      X(allowed,direction)
```

5.5.2 Verification Approach

To restrict the state space whilst maximising the behaviour covered in our analysis we have settled on the following configuration of the translation.

Optimisation Settings. Ports open to the environment are made synchronous but there is no restriction on when the environment can send a message. `inEventPool` selfloops are disabled as they are not referenced by the requirements. Four `inState` selfloops are enabled, as they are referenced by the requirements: “ALL_LEFT", “ALL_RIGHT", “WAITING_FOR_INITIALISING", “PDI_CONNECTION_ESTABLISHED".

Requirement	**Result**	**Time no counter**	**Time counterexample**
REQ_P_1	false	35m	461m
REQ_P_1_1	true	75m	
REQ_P_2	false	103m	500m

Table 5.24: Verification results for the SCI-P interface. The last two columns respectively show the time in minutes to verify a requirement with and without the counterexample generation feature. In both cases the time needed by the tools `lts2pbes` and `pbessolve` is combined.

Restrictions Configuration Ports. The configuration ports are disabled as they cannot change value during execution. We configured it to use a single point machine (D13_PM2_Activation is to "INACTIVE"). There is a configuration port for a country code which we set to the Dutch code. This means that some behaviour is disabled. In particular there is no functionality for trailed points. The timeout value for moving the point is set to a fixed value (the specific value is irrelevant as our models abstract from time).

Other Restrictions Environment Ports. There are two ports communicating the state of the generic layer to the point, which, in the point model are ports open to the environment. Many values that are sent to these ports lead to the same behaviour in the point specific layer. We could therefore have restricted what values can be sent to these ports. However, in the context of compositional minimisation, restricting these ports would not help as states that have the same behaviour with a different value for these environment ports will be bisimilar.

5.5.3 Results

Linearising, and compositionally constructing the state space took 37 minutes, resulting in an LTS with 9.5 million states. The two requirements do not hold for the model (see Table 5.24). We will go over these unsatisfied requirements and see why they do not hold.

Requirement REQ_P_1, which states that changes to the position of the point will always eventually be reported to the interlocking, does not hold. The counterexample shows that the environment introduces loops of behaviour where a message is sent and immediately discarded. A weakened requirement REQ_P_1_1, which only considers paths in which the environment does not send inputs, does hold for the model. We will revisit this requirement in Chapter 6, where we will embed a justness assumption in the formula.

For REQ_P_2 we get the following following sequence of events as counterexample.

1. The interlocking requests the object controller to move the point to the left position.

2. The interlocking requests the object controller to move the point to the right position.

3. The object controller initiates a movement to the left (the command to the right is not processed yet).

This kind of behaviour is possible due to the asynchronous communication, which is realistic for communication over an IP network.

It is not obvious which weaker requirement would hold for the model. We could maintain a list of commands from the interlocking that may lead to movements of the point. But it is not clear (by observing the outputs of the object controller) when a command is popped from that list as the object controller may consume a movement command without moving the point when it is already in the correct position. Another problem that arises is that from the perspective of the interlocking it is unclear when the object controller has processed all commands and will not initiate a new movement.

Eventually the object controller will have processed all the commands and not initiate a new movement. We have, by hand, tried to find the worst case scenario based on the observations from trying to verify the μ-calculus formula. We found the following scenario, which is depicted in Figure 5.9. Note that RaSTA has a timeout value (T_{max}); when no messages have been received in that timeframe the connection is closed. We represent time in the scenario with the variable T. We conclude that only after $2 \times T_{max}$ time after the last movement command we can assume that the object controller will not initiate a new movement. This is an extra requirement for the interlocking that should be made explicit by EULYNX, either in text or by modelling it in SysML. The point specific component of the interlocking could be adjusted to only report an end position to the core interlocking $2 \times T_{max}$ time after the last movement command was sent.

5.6 Subsystem Level Crossing Interface (SCI-LC)

The SCI-LX interface is the adjacent system interface, used, among others, by DB Netz AG. The SCI-LC interface is the subsystem variant of the level crossing interface used, among others, by ProRail. The main behaviour specified in the SCI-LX interface is message passing between the core interlocking and the level crossing protection facility outside the scope of EULYNX.

The SCI-LC interface contains a lot more behaviour than the SCI-LX interface. The SCI-LC object controller contains logic to decide when a status update is sent to the interlocking. Additionally, it can choose to ignore commands from the interlocking when it is set to the state ‘isolated’, or deactivate the level crossing when the connection with the interlocking is lost. The SysML model is then also a lot bigger than the SCI-LX SysML model.

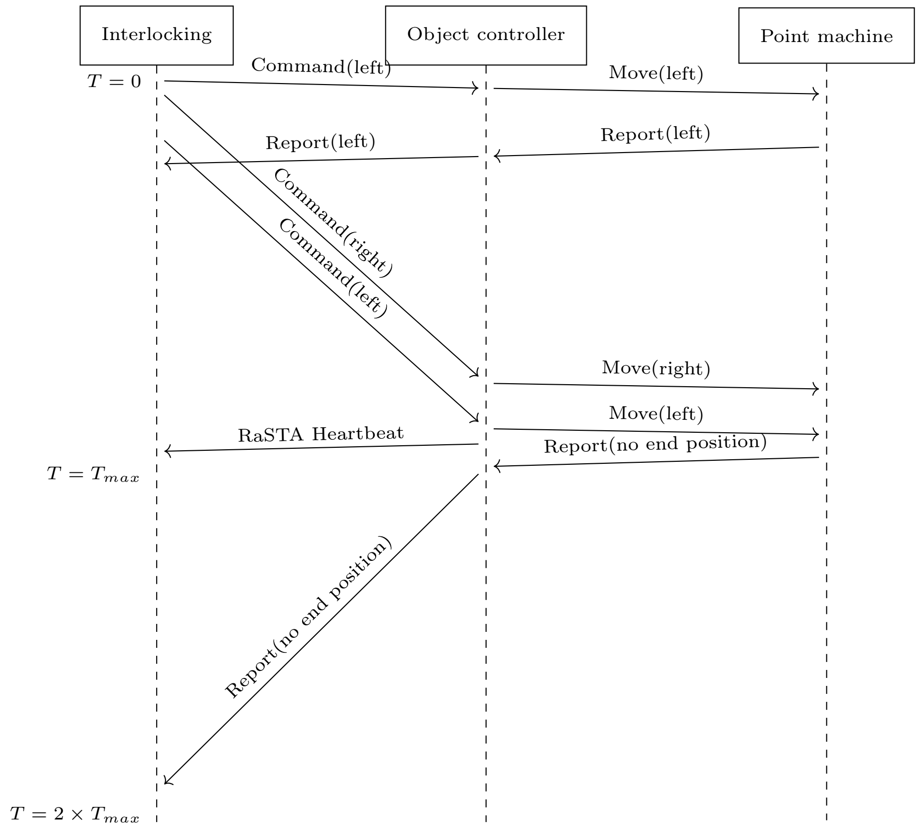


Figure 5.9: Scenario in the point interface where after the interlocking commanding the left position and the object controller reporting the left position the point can still move.

The object controller is connected to the Level Crossing Protection Facility (LCPF), which is an installation that may consist of barriers, warning bells, warning lights and sensors to detect whether the barriers are closed. The object controller can steer the LCPF to three activation levels:

1. Deactivated: warning signals stop and traffic is free to cross the tracks;
2. Pre-activated: warning signals indicate that the LCPF will soon be activated (this activation level is optional);
3. Activated: the level crossing is closing/closed; a train may pass once the the crossing is fully closed for traffic.

It can differ between countries and even between installations how the LCPF behaves when it is (pre)-activated. Typically, when it is pre-activated it will begin ringing bells and flashing lights. When it is activated it will close the barriers (if they are present) to prevent traffic from reaching the tracks. The object controller can observe two protection levels: protected and unprotected.

Division in Components. The SCI-LC specific interface is modelled using three components, see Figure 5.10. A pair of interlocking/object controller components (S_SCI_LC_SR and F_SCI_LC_SR) implement the exchange of SCI-LC specific commands and messages. Component F_LC_Functions_SR contains most of the behaviour and interacts with the LCPF. The LCPF comprises of the sensors and actuators to close the barriers, warning lights, etcetera. The LCPF is outside the scope of EULYNX. The state machine belonging to F_LC_Functions_SR contains 4 parallel regions with the following functions.

- Monitoring a closure timer. The timer is set by another parallel region when the LCPF is activated. This region then monitors whether the timer is overrun.
- Monitoring and controlling the LCPF. This parallel region contains the logic to (de)activate the LCPF as commanded by the interlocking and reports the current functional status to the interlocking.
- Status reporting. This component monitors status messages coming from the LCPF and reports them to the interlocking. These status messages include obstacle detection (if this function is supported) and failure reports.
- Handling local operations. The level crossing can be put in a 'local' mode by the interlocking. In this mode the level crossing can be controlled by a local operator. This parallel region handles the local operations and the handover of control between the interlocking and the local operator.

The state machine of F_LC_Functions_SR contains five sub state machines, making it a relatively large component.

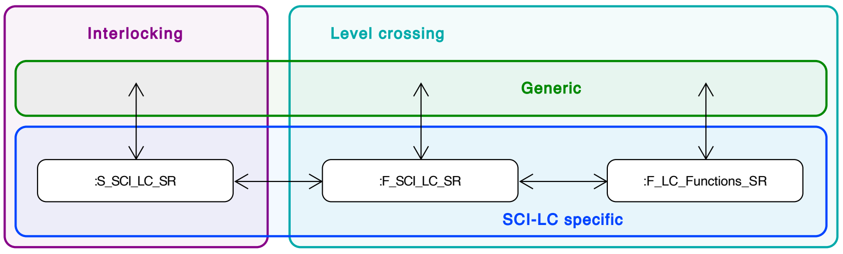


Figure 5.10: Structure of the components of the EULYNX SCI-LC interface. Arrows indicate communication channels between components.

5.6.1 Requirements

The main safety consideration with level crossings is train-vehicle and train-person collisions; this is the reason level crossing protection facilities exist in the first place. A hazardous situation arises when the LCPF is deactivated when it should be activated due to an approaching train. It can also be dangerous when the LCPF is activated for too long; at some point waiting traffic becomes impatient and might try to cross the tracks regardless of the warning signals.

As with the other EULYNX interfaces two general requirements are that the object controller should only change the state of the field element when commanded to do so by the interlocking and it should always accurately report the status of the field element. The requirements REQ_LC_1 and REQ_LC_2 make these general requirements specific to the level crossing interface.

ID	REQ_LC_1
Summary	The object controller does not change the activation level of the LCPF without a command from the interlocking.
Detailed description	The object controller will only change the activation level of the LCPF to the activation level last commanded by the interlocking. For activating the level crossing there are two exceptions: the component F_LC_Functions_-SR is in the state INITIAL_OUTPUT_STATES or IN_-STATE_PDI_CONNECTION_CLOSED. For deactivating the level crossing there is one exception: the component F_LC_Functions_SR is in the state IN_STATE_PDI_-CONNECTION_CLOSED. Note: when the PDI connection is closed the level crossing is first activated and after a set timeout deactivated again.

μ-calculus formula

```
%0 = no command, 1=activation, 2=preactivation, 3=deactivation
νX(latest_command: Int = 0).
  [send(CompPortPair(ENV_slc, T1_Realise_Activation),
      Value_Pulse_Pack([VarValuePair(DT1_Realise_Activation,
          Value_Custom(STR_Activation))]))]X(1)
  ∧ [send(CompPortPair(ENV_slc, T1_Realise_Activation),
      Value_Pulse_Pack([VarValuePair(DT1_Realise_Activation,
          Value_Custom(STR_Pree_Activation))]))]X(2)
  ∧ [send(CompPortPair(ENV_slc, T2_Realise_Deactivation),
      Value_Pulse_Pack([]))]X(3)
  ∧ [¬(∃ v:Value. send(CompPortPair(ENV_slc, T1_Realise_Activation), v)
    ∨ send(CompPortPair(ENV_slc, T2_Realise_Deactivation), v))]
      X(latest_command)
  ∧ ((val(latest_command ≠ 1)
    ∧ [inState(BEQ_functions, IN_STATE_PDI_CONNECTION_CLOSED)]⊥
    ∧ [inState(BEQ_functions, INITIAL_OUTPUT_STATES)]⊥)
    => [send(CompPortPair(BEQ_functions, T31_Activate_LCPF),
        Value_Pulse_Pack([]))]⊥)
  ∧ ((val(latest_command ≠ 2))
     => [send(CompPortPair(BEQ_functions, T33_Pre_Activate_LCPF),
         Value_Pulse_Pack([]))]⊥)
  ∧ ((val(latest_command ≠ 3)
    ∧ [inState(BEQ_functions, IN_STATE_PDI_CONNECTION_CLOSED)]⊥)
    => [send(CompPortPair(BEQ_functions, T32_Deactivate_LCPF),
        Value_Pulse_Pack([]))]⊥)
```

ID	REQ_LC_2
Summary	The object controller always accurately reports the status of the LCPF.
Detailed description	The object controller tracks the activation status of the LCPF (activated, pre-activated or deactivated). It also tracks the protection status of the LCPF (protected or unprotected). These two are combined in “Functional_-Status” reports to the interlocking, which can be • Activated and protected, • Activated and unprotected, • Deactivated and unprotected, • Or Pre-activated. Each functional status report by the component F_LC_-Functions_SR should accurately reflect the activation level and protection status of the LCPF.

μ-calculus formula

```
%0 = deactivation, 1=activation, 2=preactivation
νX(latest_command:Int = 1, protected:Bool = ⊥).
  [send(CompPortPair(BEQ_functions, T31_Activate_LCPF),
      Value_Pulse_Pack([]))]X(1, protected)
  ∧ [send(CompPortPair(BEQ_functions, T32_Deactivate_LCPF),
      Value_Pulse_Pack([]))]X(0, ⊥)
  ∧ [send(CompPortPair(BEQ_functions, T33_Pre_Activate_LCPF),
      Value_Pulse_Pack([]))]X(2, protected)
  ∧ [send(CompPortPair(ENV_functions, T30_Status_LCPF),
      Value_Pulse_Pack([VarValuePair(DT30_Status_LCPF,
          Value_Custom(STR_Proteecteed))]))]X(latest_command, ⊤)
  ∧ [send(CompPortPair(ENV_functions, T30_Status_LCPF),
      Value_Pulse_Pack([VarValuePair(DT30_Status_LCPF,
          Value_Custom(STR_Unproteecteed))]))]X(latest_command, ⊥)
  ∧ (val(latest_command ≠ 1 ∨ ¬protected)
    => [send(CompPortPair(BEQ_functions, T5_Msg_LC_Functional_Status),
        Value_Pulse_Pack([VarValuePair(
          DT105_Report_LC_Functional_Status,
          Value_Custom(STR_Activateed_and_proteecteed))]))]⊥)
  ∧ (val(latest_command ≠ 1 ∨ protected)
    => [send(CompPortPair(BEQ_functions, T5_Msg_LC_Functional_Status),
        Value_Pulse_Pack([VarValuePair(
          DT105_Report_LC_Functional_Status,
          Value_Custom(STR_Activateed_and_unproteecteed))]))]⊥)
  ∧ (val(latest_command ≠ 0)
    => [send(CompPortPair(BEQ_functions, T5_Msg_LC_Functional_Status),
        Value_Pulse_Pack([VarValuePair(
          DT105_Report_LC_Functional_Status,
          Value_Custom(STR_Deeactivateed_and_unproteecteed))]))]⊥)
  ∧ [¬((∃ p:VarName.
      val(p ∈ [T31_Activate_LCPF, T32_Deactivate_LCPF,
          T33_Pre_Activate_LCPF])
      ∧ send(CompPortPair(BEQ_functions, p), Value_Pulse_Pack([])))
    ∨ (∃ s:Custom_Value. val(s ∈ [STR_Proteecteed, STR_Unproteecteed])
      ∧ send(CompPortPair(ENV_functions, T30_Status_LCPF),
          Value_Pulse_Pack([VarValuePair(DT30_Status_LCPF,
              Value_Custom(s))]))))]X(latest_command,protected)
```

A level crossing can be *isolated* by the interlocking, after which it will no longer respond to (de)activation commands and will not activate the level crossing when the connection is lost. Only when the interlocking explicitly sends a command to de-isolate the level crossing it will operate normally again. This mode is useful when an interlocking is turned off for maintenance, in which case no trains can run so the level crossing need not be activated when the connection is lost. The requirement has a weaker variant called REQ_LC_3_1 that will be further discussed in Section 5.6.3

ID	REQ_LC_3
Summary	The object controller does not enter or leave the state ISOLATED without a command from the interlocking. When the interlocking commands to enter or leave the state ISOLATED it always eventually does so.
Detailed description	The object controller starts in a state where it is not isolated. When the object controller receives a command to isolate it must always eventually do so by setting Mem_Last_LC_State to “Isolated”. When the object controller receives a command to de-isolate it must always eventually do so by changing Mem_Last_LC_State to another value. When the generic layer signals that its state is BOOTING or NO_OPERATING_VOLTAGE then the object controller must also always eventually de-isolate itself by changing Mem_Last_LC_State to another value.

μ-calculus formula

```
νX(isolated: Bool = ⊥).
  [¬((∃ str:Custom_Value.
    val(str ∈ [STR_Isolatee_LC_eenablee, STR_Isolatee_LC_disablee])
    ∧ send(CompPortPair(BEQ_slc, T104_Cd_Isolate_LC),
        Value_Pulse_Pack([VarValuePair(DT4_Cd_Isolate_LC,
            Value_Custom(str))])))
    ∨ send(CompPortPair(ENV_functions, D50_EST_EfeS_State),
        Value_Custom(STR_BOOTING))
    ∨ send(CompPortPair(ENV_functions, D50_EST_EfeS_State),
        Value_Custom(STR_NO_OPERATING_VOLTAGE)))
  ]X(isolated)
  ∧ (val(¬isolated) => (μY.
    [¬(eventPoolFull ∨ (∃ c:CompName, s:StateName. inState(c,s))
      ∨ (∃ v:Value. varVal(BEQ_functions, Mem_Last_LC_State, v)))]Y
    ∨ [varVal(BEQ_functions, Mem_Last_LC_State,
        Value_Custom(STR_Isolateed_LC))]⊥))
  ∧ (val(isolated) => (μY. [¬(eventPoolFull
    ∨ (∃ c:CompName, s:StateName. inState(c,s))
    ∨ (∃ v:Value. varVal(BEQ_functions, Mem_Last_LC_State, v)))]Y
    ∨ <varVal(BEQ_functions, Mem_Last_LC_State,
        Value_Custom(STR_Isolateed_LC))>⊤))
  ∧ [send(CompPortPair(BEQ_slc, T104_Cd_Isolate_LC),
      Value_Pulse_Pack([VarValuePair(DT4_Cd_Isolate_LC,
          Value_Custom(STR_Isolatee_LC_eenablee))]))]X(⊤)
  ∧ [send(CompPortPair(BEQ_slc, T104_Cd_Isolate_LC),
      Value_Pulse_Pack([VarValuePair(DT4_Cd_Isolate_LC,
          Value_Custom(STR_Isolatee_LC_disablee))]))]X(⊥)
  ∧ [(send(CompPortPair(ENV_functions, D50_EST_EfeS_State),
      Value_Custom(STR_BOOTING))
    ∨ send(CompPortPair(ENV_functions, D50_EST_EfeS_State),
        Value_Custom(STR_NO_OPERATING_VOLTAGE)))
  ]X(⊥)
```

Variant ID	REQ_LC_3_1
Variant summary	The object controller does not enter or leave the state ISOLATED without a command from the interlocking. When the interlocking commands to enter or leave the state ISOLATED *while the SCI-LC specific layer is ready to handle the command* it always eventually does so.
Detailed description variant	The same as the original requirement with the difference that a command should only be handled when F_SCI_LC_SR is in the state TRANSMIT_COMMANDS_OR_MESSAGES and F_LC_Functions_SR is in the state OPERATIONAL.

μ-calculus formula

```
νX(isolated: Bool = ⊥).
  [¬((∃ str:Custom_Value.
      val(str ∈ [STR_Isolatee_LC_eenablee, STR_Isolatee_LC_disablee])
      ∧ send(CompPortPair(BEQ_slc, T104_Cd_Isolate_LC),
          Value_Pulse_Pack([VarValuePair(DT4_Cd_Isolate_LC,
            Value_Custom(str))])))
    ∨ send(CompPortPair(ENV_functions, D50_EST_EfeS_State),
        Value_Custom(STR_BOOTING))
    ∨ send(CompPortPair(ENV_functions, D50_EST_EfeS_State),
        Value_Custom(STR_NO_OPERATING_VOLTAGE)))
  ]X(isolated)
  ∧ (val(¬isolated) => (μY. [¬(eventPoolFull
      ∨ (∃ c:CompName, s:StateName. inState(c,s))
      ∨ (∃ v:Value. varVal(BEQ_functions, Mem_Last_LC_State, v)))]Y
    ∨ [varVal(BEQ_functions, Mem_Last_LC_State,
        Value_Custom(STR_Isolateed_LC))]⊥))
  ∧ (val(isolated) => (μY. [¬(eventPoolFull
      ∨ (∃ c:CompName, s:StateName. inState(c,s))
      ∨ (∃ v:Value. varVal(BEQ_functions, Mem_Last_LC_State, v)))]Y
    ∨ <varVal(BEQ_functions, Mem_Last_LC_State,
        Value_Custom(STR_Isolateed_LC))>⊤))
  ∧ ((<inState(BEQ_flc, TRANSMIT_COMMANDS_OR_MESSAGES)>⊤
    ∧ <inState(BEQ_functions, OPERATIONAL)>⊤
    ∧ <inState(BEQ_functions, DEACTIVATED)>⊤
  ) => [send(CompPortPair(BEQ_slc, T104_Cd_Isolate_LC),
      Value_Pulse_Pack([VarValuePair(DT4_Cd_Isolate_LC,
          Value_Custom(STR_Isolatee_LC_eenablee))]))]X(⊤))
  ∧ ((<inState(BEQ_flc, TRANSMIT_COMMANDS_OR_MESSAGES)>⊤
    ∧ <inState(BEQ_functions, OPERATIONAL)>⊤
  ) => [send(CompPortPair(BEQ_slc, T104_Cd_Isolate_LC),
      Value_Pulse_Pack([VarValuePair(DT4_Cd_Isolate_LC,
          Value_Custom(STR_Isolatee_LC_disablee))]))]X(⊥))
  ∧ (¬(<inState(BEQ_flc, TRANSMIT_COMMANDS_OR_MESSAGES)>⊤
    ∧ <inState(BEQ_functions, OPERATIONAL)>⊤
    ∨ ([inState(BEQ_functions, DEACTIVATED)]⊥ ∧ val(¬isolated))
  ) => [(∃ v:Value. send(CompPortPair(BEQ_slc, T104_Cd_Isolate_LC), v))]
    X(isolated))
  ∧ [(send(CompPortPair(ENV_functions, D50_EST_EfeS_State),
      Value_Custom(STR_BOOTING))
    ∨ send(CompPortPair(ENV_functions, D50_EST_EfeS_State),
        Value_Custom(STR_NO_OPERATING_VOLTAGE)))]X(⊥)
```

Requirement	Result	Time no counter	Time counterexample
REQ_LC_1	true	10m	
REQ_LC_2	false	10m	49m
REQ_LC_3	false	13m	71m
REQ_LC_3_1	true	19m	

Table 5.29: Verification results for the SCI-LC interface. The last two columns respectively show the time in minutes to verify a requirement with and without the counterexample generation feature. In both cases the time needed by the tools `lts2pbes` and `pbessolve` is combined.

5.6.2 Verification Approach

To restrict the state space whilst maximising the behaviour covered in our analysis we have settled on the following configuration of the translation.

Optimisation Settings. Ports open to the environment are restricted by enforcing that there is no event in the event queue of any of the components. This strong restriction is needed to reduce the state space to such a level that verification is feasible. Five `inState` selfloops are enabled, as they are referenced by the requirements: "IN_STATE_PDI_CONNECTION_CLOSED", "INITIAL_OUTPUT_-STATES", "ISOLATED", "OPERATIONAL", and "TRANSMIT_COMMANDS_-OR_MESSAGES". One `varVal` loop is enabled for Mem_Last_LC_State.

Restrictions Configuration Ports. The configuration ports are disabled as they cannot change value during execution. We configured them to enable a closure timer, connection loss timer, isolation and pre-activation. We have disabled obstacle detection. Timeout values are set to a fixed value; the specific value is irrelevant as our models abstract from time.

5.6.3 Results

Linearising, and compositionally constructing the state space took 103 minutes, resulting in an LTS with 2.7 million states. Verifying the requirements (in parallel) without counterexample generation took 22 minutes. The results are summarised in Table 5.29.

Requirement REQ_LC_1 holds for the model but likely only because of the optimisation options that we selected, in particular due to the restriction that the environment only sends a message when there is no event in any event queue. We expect the same issues as we saw with the point. When the interlocking sends multiple conflicting commands to the field element and the first and last command are the same, then it is unclear whether the object controller has processed all commands or may still change the status of the field element by processing the remaining commands.

The counterexample for requirement REQ_LC_2 reveals that the object controller can report an inaccurate status of the LCPF, a serious issue. By examining the SysML model with the counterexample in mind we discovered that it can report 'Activated and protected' whilst it is actually 'Deactivated and unprotected' and vice versa.

The problem lies in the fact that in multiple places (de)activating the LCPF and setting a local variable 'Mem_Last_LC_State' happens in two different states with a SysML transition (without guard or trigger in between), see Figure 5.11. This transition may never be taken since, as an alternative, a transition may be fired on higher level when the PDI connection is lost/re-established. The following sequence of events then becomes possible.

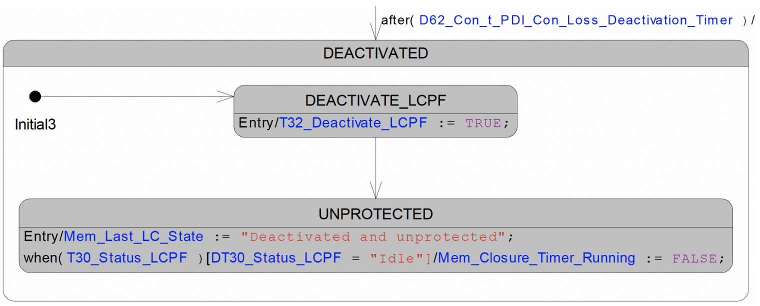

after(D62_Con_t_PDI_Con_Loss_Deactivation_Timer) /
DEACTIVATED
Initial3
DEACTIVATE_LCPF
Entry/T32_Deactivate_LCPF := TRUE;
UNPROTECTED
Entry/Mem_Last_LC_State := "Deactivated and unprotected";
when(T30_Status_LCPF)[DT30_Status_LCPF = "Idle"]/Mem_Closure_Timer_Running := FALSE;

Figure 5.11: Fragment of the state machine belonging to the component F_LC_Functions_SR.

1. The connection between the interlocking is lost and the object controller has as a result activated the LCPF. The variable Mem_Last_LC_State is set to "Activated and protected".
2. The connection loss deactivation timer expires.
3. The object controller moves to the state DEACTIVATE_LCPF (as seen in Figure 5.11) and deactivates the LCPF. Note that Mem_Last_LC_State has not yet been updated at this point.
4. The connection with the interlocking is re-established.
5. The object controller moves to the state OPERATIONAL. Since Mem_-Last_LC_State is still set to "Activated and protected" it moves to the substate PROTECTED (without reactivating the LCPF).
6. The object controller reports the functional states "Activated and protected" to the interlocking.

The final requirement also does not hold for the model. The counterexample generated by the toolset shows that a command to (de)isolate the level crossing can be discarded when the command arrives at the object controller when it is not (yet) ready to process it because the object controller blocks specific to the level crossing are still in the state for establishing a connection with the interlocking. One might think that such a run is unrealistic: how can a command already be sent when one side is still in the process of establishing the connection? However, the following scenario is allowed by the model due to the subdivision in blocks.

1. Starting point: the connection between the object controller and the interlocking is not established.
2. They establish a connection. The generic blocks are now in a state reflecting that the connection is established but the blocks specific to the level crossing are not (yet).
3. At the interlocking side the level crossing specific layer is notified of the established connection.
4. The interlocking sends a command to (de)isolate the level crossing.
5. The object controller discards this message in the level crossing specific layer as it is not yet ready to process the command.
6. The level crossing specific layer is notified that the connection is established but the message was already discarded.

This scenario is unlikely (but not impossible) in practice because the generic blocks would notify the level crossing specific blocks in a fraction of a second. An adjusted requirement, REQ_LC_3_1, which allows discarding of commands in this scenario, does hold for the model.

5.7 Limitations and Threats to Validity

Model checking allows for exhaustive verification of the model, in our case the mCRL2 model derived from the EULYNX SysML models. This does not guarantee that there are no errors in the EULYNX specification, even if all relevant requirements are identified and pass verification. The verified mCRL2 model does not cover everything from the EULYNX specifications. Some things are only documented in text and are not modelled in SysML, such as the bit-level encoding of EULYNX messages; they are outside the scope of analysis.

The mCRL2 model could not account for all such details even if we added them manually; some level of abstraction is necessary to keep the size of the induced state space manageable. For example, integer overflows and whether byte encoding is big endian or little endian need to be abstracted from. Errors or underspecification on these fronts will not be caught by model checking. This is not necessarily a downside of our approach, it is a common engineering practice to use abstraction to reason about higher level designs.

A specific concern in FormaSig is that the mCRL2 model is more precise than the SysML model, and at some points less permissive. We have tried to take the most permissive interpretation of the SysML models but this is not always possible due to the state space explosion problem. We need to keep the size of buffers very small and do not allow all interleavings in ASAL execution, making the model more restrictive. As a result, the SysML model, which is the authoritative model, allows for more behaviour than the mCRL2 model. Hence, our analysis does not cover all allowed behaviour.

Besides these limitations we should also ask ourselves whether we can trust the outcomes of the tools at all: can bugs create false positives or false negatives in the verification results? We can identify the following possible points of failure:

1. The translation from SysML to mCRL2;
2. The encoding of SysML models in our framework;
3. The formalisation of requirements in the modal μ-calculus;
4. The mCRL2 toolset.

For all points of failure it holds that false negative outcomes of model checking are not very severe, although they can be cumbersome to identify. If a μ-calculus formula evaluates to false, we simply check the counterexample provided by the tools and manually check whether we can replay the scenario on the original SysML model.

1. Translation from SysML to mCRL2. Establishing that the translation from SysML to mCRL2 is fully correct is difficult due to the complexity of the resulting mCRL2 model. At the moment we verify the mCRL2 model produced by our translation framework by stepping through a part of the model using `lpsxsim`, checking whether the behaviour is as intended. Additionally, we verify that all SysML states are reachable using a μ-calculus formula.

On a more fundamental level, correctness of the translation cannot be mathematically established without a formal semantics of SysML independent from our formalisation in mCRL2. In our approach the translation *is* the formalisation. We can merely assess whether the formalisation in mCRL2 and the translation tool `sysml2mcrl2` work as intended.

2. Encoding the SysML Models. The measures taken to check the translation framework will also catch errors in the encoding of the SysML models. Moreover, the translation framework itself performs a number of sanity checks, such as whether state machine diagrams contain states that are not connected to the initial state via transitions, or states without outgoing transitions.

Model	**#States**	**State space generation**	**Time no counter**	**Time counterexample**
SCI-LX	16.8×10^6	91m	789m	1,496m
Generic	12.6×10^6	50m	118m	998m
SCI-P	9.5×10^6	37m	103m	500m
SCI-LC	2.6×10^6	103m	19m	71m

Table 5.30: Summary of the verification effort for each interface. The last two columns respectively show the maximum time in minutes to verify a requirement with and without the counterexample generation feature. As the verifications are performed in parallel the longest running time is the bottleneck for the total verification time.

3. Formalization of Requirements. A risk in formulating (formal) requirements is that they may not express what you intended. A formula may even be trivially true or false due to a mistake in the formulation. We have tested the correctness of some formulas by injecting a fault in the model corresponding to what the requirement expresses, with the expectation that the formula becomes false. This adds a layer of redundancy, improving confidence in the results [22]. Future work, which will be discussed in the next section, includes research towards formal (visual) requirements languages that can be understood by signalling engineers, allowing them to validate the formalised requirements.

4. mCRL2 Toolset. The correctness of our findings relies on the correctness of the mCRL2 toolset. Verifying the correctness of the mCRL2 toolset is outside the scope of this thesis.

5.8 Conclusion

The approach to scalability as described in the previous chapter has proven to be effective: we are able to verify requirements for EULYNX models in a reasonable time frame but not on an ordinary office machine due to the amount of RAM required. Since the optimisations that alter the semantics are optional, we can evaluate more paths of the system for smaller EULYNX interfaces, whilst being able to handle the larger interfaces. Table 5.30 provides a summary of the time it took to run the verifications.

Compared to an earlier case study of the point interface [21], which was performed in 2021, the performance of our toolchain has improved a lot. Back then, we were forced to use the `pbessolvesymbolic` tool to verify requirements, which is currently not able to produce counterexamples. Moreover, even with the symbolic tool, we were not able to verify whether requirement REQ_P_2 holds for the model. Now, we are able to check all requirements with counterexample generation within a day.

The optimisations that we use can alter the outcome of model checking. This begs the question what conclusions we can draw from the verification outcomes. If a formula does not hold we can inspect a counterexample and examine whether there is indeed a flaw in the model. Hence, the activity of model checking improves the *quality* of the model. If a formula does hold we cannot consider it to be a definitive proof as some runs of the systems have been excluded. It does, however, increase the *confidence* in the correctness of the system as more runs of the system are analysed than a human ever could and without the possibility of human error.

Analysing the specification from different perspectives and/or abstraction levels can help to uncover more issues. The issues with RaSTA were discovered in the SCI-LX case study. Before, we had already performed a case study of the generic subsystem interface [20, 21]. In this earlier case study we removed the RaSTA components (RaSTA was made part of the environment) to reduce the state space. This was not necessary for the SCI-LX case study allowing us to analyse the interaction with RaSTA. In hindsight, we could have discovered the issue with RaSTA already in the generic subsystem case study by creating a second model in which RaSTA was included and the components that do not directly interact with RaSTA were not.

An aspect that has not received much attention in this chapter is that obtaining the right requirements is a challenging task in which our iterative approach was crucial; the requirements and models are too subtle to use a waterfall workflow, feedback loops are necessary. For many requirements it took a number of iterations to formalise the requirement correctly and cover all corner cases.

In EULYNX the object controller and interlocking are broken down into multiple components that handle a specific function. This way of modelling increases the state space. Moreover, a number of the requirements failed due to reasons related to asynchronous communication between components and how events are scheduled. EULYNX should critically evaluate whether asynchronicity is desirable within the object controller and interlocking, especially since (PLC) implementations will be much more deterministic.

Communication between the interlocking and the object controller will always be asynchronous in nature. As we have seen with REQ_P_2 and REQ_LC_1 this can lead to issues. As discussed in Section 5.5.3 these issues could be mitigated by letting the interlocking wait for some time until it is ensured that all commands are processed. An alternative mitigation approach is to add sequence numbers to the commands from the interlocking. When the object controller processes a command it could report it back to the interlocking with an acknowledgement message, making it transparent for the interlocking when all movement commands have been processed. Since RaSTA already implements sequence numbers it could also be specified that RaSTA acknowledges a sequence number when the message has been processed, instead of when the message has arrived in the queue as the RaSTA specification dictates.

The case studies have proven useful for both the academic and industrial stakeholders of FormaSig. The railway infrastructure managers have been provided with valuable feedback on errors and omissions in their specifications. From the academic perspective it was useful to evaluate how well our tools and techniques could handle real specifications. In fact most of the scalability measures we have, including compositional branching bisimulation minimisation, have been developed with the purpose of being able to handle a EULYNX specification of which the state space was at times too large to handle.

Chapter 6

Verifying Liveness Requirements for Just Paths

He guards the paths of the just and protects those who are faithful to Him.

Proverbs 2:8

Every process-algebraic specification of a distributed algorithm or system includes unrealistic finite or infinite computations in which some component never makes progress [33]. Since such unrealistic computations typically violate liveness properties, their mere existence is in the way of a proof that all realistic computations do satisfy these properties. Unrealistic computations are then often excluded from consideration by imposing additional assumptions such as *progress* and (strong) *fairness*. See [49] for a comprehensive overview of such assumptions.

Progress states that when at least one action is enabled, one of them will eventually be taken. The system will not just stall. As an example, the process expression $a.P$ will always perform an a action. Without assuming progress any liveness property stating that always eventually some action is performed would not hold. Progress is a natural assumption and is typically made implicitly.

As we have seen in Chapter 5, progress is not sufficient for some of the requirements we want to verify. In the counterexample for requirement REQ_PDI_6 we saw that the requirement does not hold due to a loop of behaviour of other components.

(Strong) fairness is a much stronger assumption than progress. An infinite path is strongly fair if every action that is infinitely often enabled is eventually taken. As an example, a process P with $P \overset{\text{def}}{=} a.P + b.P$ will always perform both a and b actions. Fairness is an assumption we are not willing to make for mCRL2 models derived from SysML models. If a state in a state machine has multiple enabled transitions we also want to consider the case where it always takes only one of the transitions. As EULYNX specifications do not specify how events are scheduled we do not want to make assumptions that are too strong. An extra consideration is that an implementation will likely be more deterministic than the model.

Van Glabbeek and Höfner have proposed *justness* [48] as a criterion that is just strong enough to exclude many unrealistic computations, but not too strong:

> "Once a transition is enabled that stems from a set of parallel components, one (or more) of these components eventually partake in a transition." [49]

This is an assumption that we consider to be safe to make in our context. Every state machine should be able to make progress.

The question then arises how we can incorporate such a justness assumption in the verification of liveness requirements (in the context of mCRL2). A challenge is that not every participation in a transition truly affects a component. For example, the `inState` selfloops in our models cannot be regarded as an action of the component, otherwise any (liveness violating) infinite path can be made just by inserting some `inState` transitions. A similar situation arises when modelling shared variables. In a process-algebraic specification shared variables are components (processes) themselves, and hence reading the value of a shared variable is modelled as an interaction of the component that reads and the component that models the variable. Hence, an unrealistic infinite computation in which one component continuously wants to assign a new value to the variable, but never actually does, can, nevertheless, be just because another component time and again reads the value of the variable.

Recent work by Van Glabbeek and coauthors suggests that the liveness property for Peterson's mutual exclusion algorithm [104], stating that any process that wants to enter the critical section will eventually enter it, cannot be analysed in CCS and related formalisms [33, 48] due to this contention of reading and writing variables. To counteract this problem, it is proposed in [33] to extend the syntax and semantics of CCS with a so-called *signal emission operator*, providing an alternative mechanism to communicate information about the state of a component (e.g., a variable) to other components. Although adding this operator does not increase the absolute expressiveness of the calculus, it does facilitate a refined definition of justness. In this refined definition, the reading of a signal is given special treatment by which computations such as the one described above are not considered just, and thus excluded from consideration.

Our contributions

The signal emission operator is a non-standard process-algebraic construction. It is not part of the specification formalism of mCRL2, nor, to the best of our knowledge, of the specification formalism of any other process-algebra based automated verification tool. The question arises whether the addition of such an operator is essential. If so, a non-trivial overhaul of established verification tools is called for. Our first contribution is to show that it is not, if one is willing to pay a small price:

there is no general formal definition of justness for the entire calculus; the formal definition must be tuned to the process expression under consideration. When aiming for an automated verification, this is indeed a negligible price, since one is just interested in the process expression that models the system under verification.

Semantically, the signal emission operator simply adds a self-loop labelled with a *signal* to the state representing the process expression to which it is applied. A signal is just a special type of label, so the self-loop can easily be specified by other means (e.g., using recursion) if a particular subset of the set of labels is designated as signals. Because the choice of an appropriate set of signals depends on how those labels are used in the process expression at hand, the formal definition of justness needs to be specific for a particular process expression.

In the absence of tools supporting the verification under justness of specifications such as Peterson's algorithm, establishing that a specification meets a property remains a manual activity. This is problematic, as the complexity of a typical specification easily leads to cases being missed in the analysis. Therefore, to conduct a convincing automated verification of a property of an algorithm, we not only need to specify the algorithm in a process-algebra based formalism; we also need to formulate the property in a suitable modal logic. Moreover, in the verification of the property, justness has to be taken into account. It is unclear, however, whether this can be achieved without changing the verification algorithms that are used to evaluate the validity of a modal-logic formula with respect to the labelled transition system associated with the process expression. A complication is, for instance, that the definition of justness refers to a notion of *component*, which naturally exists at the level of the syntactic representation of the system (i.e., the process expression), but not at the labelled transition-system level.

Our second contribution is derived from the observation that with the ACP-style communication mechanism [6] of mCRL2, which is more general than the communication mechanism of CCS, Peterson's algorithm can be specified in such a way that justness can be defined referring to labels rather than to components. The idea is to achieve a partitioning of the set of labels that reflects the component structure of the process expression. It is then possible to reformulate justness referring to labels, rather than to components. We generalise the observation regarding Peterson's algorithm and formulate general syntactic conditions that ensure that such a partitioning is possible.

Our third contribution is a template modal μ-calculus formula that expresses a typical liveness property, asserting that on all just paths, an action, say a, is eventually followed by another, say b. This template formula can easily be instantiated by a user wishing to carry out a liveness verification of an algorithm, and only requires information concerning which actions are designated as signals. As a result, standard, off-the-shelf tooling such as mCRL2 can be used to automatically verify liveness properties of algorithms such as Peterson's mutual exclusion protocol. In case such verifications fail, evidence can be provided [30, 116], helping the user to pinpoint the root cause.

The final contribution is an application in the verification of EULYNX interfaces. We show in a general way how liveness requirements can be verified with the mCRL2 toolset for models which are an instantiation of the generic mCRL2 model as presented in Chapter 3. We also revisit two liveness requirements from Chapter 5 that did not hold. During the case studies we already suspected that a justness or fairness assumption might make them true. It turns out that for these two requirements a justness assumptions is sufficient.

This chapter is organised as follows. In Section 6.1, following [47], we take the notion of labelled transition system with concurrency (LTSC) as technical starting point, and present a definition of justness for it. In Section 6.2 we present a process calculus that is very similar to CCS, except that it has the more general ACP-style communication mechanism. Inspired by the LTSC-semantics that van Glabbeek gives for CCS and its extension with signals in [47], we propose an LTSC-semantics for the process calculus. Then, in Section 6.3 we recapitulate in more detail the argument presented in [33] that Peterson's algorithm cannot be rendered in the process calculus in such a way that all unrealistic paths are excluded by assuming justness. In Section 6.4 we then include a semantic treatment of special labels that take the role of signals. In Section 6.5, we define when an LTSC admits a label-based treatment of justness, proposing a subclass of LTSCs that have a *concurrency-consistent labelling.* In Section 6.6, we present sufficient conditions on process expressions ensuring that the associated LTSC has a concurrency-consistent labelling. Process expressions satisfying these syntactic conditions are amenable to verifications that take justness into account. In Section 6.7 we formalise a general liveness property under justness assumptions for an LTSC that has a concurrency-consistent labelling. In Section 6.8 we comment on the actual verification of the liveness property for Peterson's algorithm with the mCRL2 toolset. In Section 6.9 we treat the application of liveness verification with a justness assumption in FormaSig. In Section 6.10 we present some conclusions.

6.1 Justness

We recap the definition of labelled transition system with concurrency and the associated notion of just path from [47].

We presuppose disjoint sets $\mathcal{A}$ and $\mathcal{S}$ of *actions* and *signals*, respectively, and let $\mathcal{L} = \mathcal{A} \cup \mathcal{S}$. Elements of $\mathcal{L}$ are generally referred to as *labels*. A *labelled transition system* (LTS) is a tuple $(\mathit{St}, \mathit{Tr}, \mathit{src}, \mathit{target}, \ell)$ with St and Tr sets of *states* and *transitions*, respectively, $\mathit{src}, \mathit{target} : \mathit{Tr} \to \mathit{St}$ and $\ell : \mathit{Tr} \to \mathcal{L}$. Note that this is an alternative notation for an LTS compared to Definition 2.1 and contains the same information. We need this alternative notation to refer to transitions explicitly.

We call a transition $t \in \mathit{Tr}$ a *signal transition* if its label is a signal and it does not change state, i.e., if $\ell(t) \in \mathcal{S}$ and $\mathit{src}(t) = \mathit{target}(t)$; otherwise, t is called an *action transition.*

Remark 6.1. *Van Glabbeek mentions in [47] that signal transitions are not supposed to change state, but does not include it as an explicit requirement. Rather, in his work, it is a consequence of the operational semantics of the process calculi under consideration that transitions labelled with signals indeed never change state.*

The syntax and operational semantics of our process calculus admit, by design, process specifications that give rise to transitions labelled with signals that do change state. We prefer that such transitions are not treated as signal transitions in the notion of justness. To this end it is convenient to include the requirement explicitly.

Signal transitions are disregarded in the definition of the notion of path. A *path* in a transition system $(St, Tr, src, target, \ell)$ is a finite or infinite alternating sequence $s_0 t_1 s_1 t_2 s_2 \cdots$ of states and action transitions, starting with a state and if it is finite also ending with a state, satisfying $src(t_i) = s_{i-1}$ and $target(t_i) = s_i$ for all relevant i. We say that a state s' is *reachable* from a state s if there exists a path that starts with s and ends with s'. We say that a transition t is *reachable* from a state s if, and only if, there is a state s' that is reachable from s and $src(t) = s'$.

Labelled transition systems abstract entirely from the notion of component. For the definition of justness, the notion of component is relevant, at least to the extent that it should be possible to determine that, whenever some transition is enabled, eventually the component (or set of components) from which the transition stems, makes progress. For the formalisation of justness, it turns out to be sufficient to consider labelled transition systems enriched with a concurrency relation on transitions [47]. We first give the formal definition of labelled transition system with concurrency; the requirements on the concurrency relation are explained after the definition.

Definition 6.2. *A* labelled transition system with concurrency *(LTSC) is a tuple $(St, Tr, src, target, \ell, \smile\!\!\bullet)$ consisting of an LTS $(St, Tr, src, target, \ell)$ and a* concurrency relation $\smile\!\!\bullet \subseteq Tr \times Tr$ *such that*

1. *$\smile\!\!\bullet$ is irreflexive on action transitions (i.e., if t is an action transition, then $t \not\smile\!\!\bullet\, t$), and*
2. *if t is an action transition and π is a path from $src(t)$ to $s \in St$ such that $t \smile\!\!\bullet\, v$ for all transitions v occurring on π, then there is an action transition u such that $src(u) = s$, $\ell(u) = \ell(t)$ and $t \not\smile\!\!\bullet\, u$.*

Intuitively, transitions are *concurrent* if they stem from different (sets of) components, and they *interfere* if they have a component in common. It is then natural to require that the concurrency relation on transitions is irreflexive: a transition cannot be concurrent with itself. Furthermore, if some component (or set of components) can perform some activity, represented by a transition t in the labelled transition system, then after executing transitions concurrent with t—which, by assumption, then stem from different components than t—it should still be possible for the component to perform that same activity. The activity can be represented by a different transition u in the labelled transition system, but this transition should not be concurrent with t (it should interfere with t, i.e., $t \not\smile\!\!\bullet\, u$) and should have the same label.

As explained in [47], justness is a *completeness criterion*: it is used to specify which paths should be considered representing a complete computation of the system. For completeness one wants to distinguish between so-called *blocking* actions and *non-blocking* actions. Intuitively, a blocking action is not entirely under the control of the system that is being specified; it may depend on interaction with the environment. A non-blocking action is thought to be completely under control of the system. A complete computation may end in a state in which only blocking actions are enabled, but not in a state in which non-blocking actions are enabled. The definition of justness takes a set of blocking actions as parameter.

Definition 6.3. *Let $\mathcal{B} \subseteq \mathcal{A}$ be a set of* blocking actions*. A path π in an LTSC is* $\mathcal{B}$-just *if for every action transition t with $\ell(t) \notin \mathcal{B}$ and $src(t) \in \pi$, a transition u occurs in the suffix of π starting at $src(t)$ such that $t \not\rightsquigarrow^{\bullet} u$.*

The example below illustrates the concept of justness.

Example 6.4. *Consider a situation in which Alice drinks coffee and eats a croissant in a small cafe, and Bob is engaged in a series of phone calls. The situation can be modelled by the following LTSC:*

$$\begin{tikzcd}
s_0 \arrow[loop above, "phone"] \arrow[r, "coffee"'] & s_1 \arrow[loop above, "phone"] \arrow[r, "croissant"'] & s_2 \arrow[loop above, "phone"]
\end{tikzcd}$$

Suppose that all labels in the above LTSC are non-blocking actions. In case all actions only interfere with themselves, the infinite path consisting of only phone transitions from state s_0 is not $\emptyset$-just since the coffee transition is enabled in s_0 but no interfering transition is ever taken on this path. In case the phone transitions in s_0, s_1 and s_2 do *interfere with the coffee transition and the croissant transition—for instance because Bob is also the waiter who serves Alice, preferring to make phone calls instead of taking her orders—then the same infinite path* is *$\emptyset$-just.*

6.2 Process Calculus

In [33], the authors claim that information exchanged through signals is essential for the characterisation of just paths in the context of Peterson's mutual exclusion algorithm; without signals, paths representing unrealistic executions of Peterson's algorithm are considered just. In [33], justifications for the claim are presented in the context of CCS. First, a version of CCS without signals is considered, Peterson's algorithm is modelled, and then it is shown that justness does not exclude all unrealistic computations. Then, Peterson's algorithm is modelled in a variant of CCS with signals, and it is shown that the corresponding notion of justness works well for Peterson's algorithm. We retrace their steps and in this section introduce a very simple process calculus to specify LTSCs that, as we show in the next section, indeed illustrates the phenomenon observed by the authors. In Section 6.4, we shall also introduce signals, but without changing the syntax of the calculus.

A special feature of our calculus, compared to CCS as considered in [33, 47], is that it includes an ACP-style communication mechanism [6]: We presuppose a binary *communication function* on the set of labels $\mathcal{L}$, i.e., a partial function

$$\gamma : \mathcal{L} \times \mathcal{L} \rightharpoonup \mathcal{L}$$

that is

- *commutative*: $\gamma(\lambda_1, \lambda_2)$ is defined if, and only if, $\gamma(\lambda_2, \lambda_1)$ is defined, and if both are defined, then we have $\gamma(\lambda_1, \lambda_2) = \gamma(\lambda_2, \lambda_1)$; and
- *associative*: $\gamma(\lambda_1, \gamma(\lambda_2, \lambda_3))$ is defined if, and only if, $\gamma(\gamma(\lambda_1, \lambda_2), \lambda_3)$ is defined, and if both are defined then $\gamma(\lambda_1, \gamma(\lambda_2, \lambda_3)) = \gamma(\gamma(\lambda_1, \lambda_2), \lambda_3)$.

This communication function defines which actions may communicate, and what is the result of that communication. Thus, communication transitions are not all labelled with the same action, as they are in CCS (in CCS all transitions that are the result of communications are labelled with τ). The advantage is that transitions that involve multiple components can be labelled such that from the label it can be determined which components are involved.

We proceed to introduce the syntax of our process calculus and associate an LTSC with it. The LTSC we get is in line with the LTSC that Van Glabbeek associates with CCS in [47], though our way of defining it deviates somewhat from van Glabbeek's in [47], as we shall explain below. For now, we presuppose that the set of signals is empty, i.e., $\mathcal{L} = \mathcal{A}$. (In Section 6.4, we shall consider the general case in which the set of signals $\mathcal{S}$ is not empty and adapt the structural operational semantics accordingly.) For the purpose of recursion, we also presuppose a set $\mathcal{I}$ of *agent identifiers*. The set $\mathcal{P}$ of *process expressions* is generated by the following grammar, with A ranging over $\mathcal{I}$, λ ranging over $\mathcal{L}$, and H ranging over subsets of $\mathcal{L}$:

$$P, Q ::= \mathbf{0} \mid \lambda.P \mid P + Q \mid P \parallel Q \mid \partial_H(P) \mid A \ . \tag{6.1}$$

The constructs $\mathbf{0}$, $\lambda.$ and $+$ are familiar from basic CCS, respectively denoting inaction, action prefix and non-deterministic choice. The construct $\parallel$ stands for ACP-style parallel composition. It represents the arbitrary interleaving of the behaviour of its components, and additionally allows its components to execute communication steps in accordance with the communication function γ: if the left component of the parallel composition can execute label λ_1 and the right component can execute label λ_2 and $\gamma(\lambda_1, \lambda_2)$ is defined, then the parallel composition can execute $\gamma(\lambda_1, \lambda_2)$. The process calculus includes encapsulation operator ∂_H (similar to the restriction operator in CCS) by which the execution of certain labels can be blocked, and thus communication between components can be enforced. The behaviour of the agent identifiers is defined through a *recursive specification* E, which is a set of defining equations

$$A \overset{\text{def}}{=} P \ ,$$

with P a process expression, including precisely one such equation for every $A \in \mathcal{I}$.

We now proceed to associate an LTSC $(\mathit{St}, \mathit{Tr}, \mathit{src}, \mathit{target}, \ell, \smile^{\bullet})$ with our process calculus. The set of states St of this LTSC is the set of process expressions $\mathcal{P}$, as usual. To define a suitable set Tr of transitions, as in [47], we take the collection of derivations in a formal proof system based on the structural operational semantics of the process calculus. We deviate from [47] in how we define the concurrency relation. In [47], Van Glabbeek inductively associates a set of *synchrons* with a derivation, which can be thought of as extracting from the derivation all the required component information necessary to define a concurrency relation. We prefer to annotate the transition relation defined by the structural operational semantics with component information directly.

First, we associate with a process expression P its *static component architecture*, which is determined by the top-level occurrences of $\|$ and ∂_H in P. Let $\mathcal{C} = \{\textsc{l}, \textsc{r}\}$; we shall refer to a component in a process expression P as a sequence in $\mathcal{C}^*$ (the empty sequence will be denoted by ϵ). We recursively associate with every process expression P a set of *components* $\mathcal{C}(P) \subseteq \mathcal{C}^*$ as follows:

- if $P = \mathbf{0}$, $P = \lambda.P'$ for some $\lambda \in \mathcal{L}$, $P = P_1 + P_2$, or $P = A$ for some $A \in \mathcal{I}$, then $\mathcal{C}(P) = \{\epsilon\}$;
- $\mathcal{C}(P_1 \| P_2) = \textsc{l} \rhd \mathcal{C}(P_1) \cup \textsc{r} \rhd \mathcal{C}(P_2)$, and $\mathcal{C}(\partial_H(P)) = \mathcal{C}(P)$.

If $X \subseteq \mathcal{C}^*$, then $\textsc{l} \rhd X = \{\textsc{l}\sigma \mid \sigma \in X\}$ and $\textsc{r} \rhd X = \{\textsc{r}\sigma \mid \sigma \in X\}$. Note that every $\sigma \in \mathcal{C}(P)$ uniquely identifies a component of P: we denote this component by $P|_{\sigma}$.

We keep track of which components contribute to a transition in the structural operational semantics for our process calculus, presented in Table 6.1. It defines a transition relation $\xrightarrow{\lambda,\alpha}$ on process expressions, which is not only endowed with a label $\lambda \in \mathcal{L}$, but also with a set $\alpha \subseteq \mathcal{C}^*$ of components.
The rule (Pref) expresses that a prefix $\lambda.P$ can do a λ-labelled transition to P; furthermore, $\lambda.P$ is by itself a component. So the set of components associated with the transition is $\{\epsilon\}$. The rules (Sum-l) and (Sum-r) express that a non-deterministic choice $P + Q$ can execute a λ-labelled transition from P or from Q. Also $P + Q$ is by itself a component, denoted by ϵ. So the set of components associated with the transition is $\{\epsilon\}$. The rule (Rec) expresses that a process name A with defining equation $A \overset{\text{def}}{=} P$ can perform any transition P can make. Also A is by itself a component, denoted by ϵ.

The rules (Par-l), (Par-r) and (Comm) express, respectively, that a parallel composition $P \| Q$ can execute a transition of the components of P, a transition of the components of Q, or execute a transition in which both components of P and Q are involved. In the latter case, the communication function γ must be defined on the labels of the transitions of P and Q and the combined transition is labelled with the result of applying the communication function to these labels. In the case of an application of (Par-l) or (Par-r), the sets of components involved in the resulting transitions need to be updated by prefixing all components suitably with $\textsc{l}$ or $\textsc{r}$, respectively. In the case of an application of (Comm), the involved components of P are prefixed with $\textsc{l}$, and the involved components of Q are prefixed with $\textsc{r}$.

$$(\textsc{Pref})\ \frac{}{\lambda.P \xrightarrow{\lambda,\{\epsilon\}} P} \qquad (\textsc{Rec})\ \frac{P \xrightarrow{\lambda,\alpha} P' \quad A \overset{\mathrm{def}}{=} P}{A \xrightarrow{\lambda,\{\epsilon\}} P'}$$

$$(\textsc{Sum-l})\ \frac{P \xrightarrow{\lambda,\alpha} P'}{P + Q \xrightarrow{\lambda,\{\epsilon\}} P'} \qquad (\textsc{Sum-r})\ \frac{Q \xrightarrow{\lambda,\alpha} Q'}{P + Q \xrightarrow{\lambda,\{\epsilon\}} Q'}$$

$$(\textsc{Par-l})\ \frac{P \xrightarrow{\lambda,\alpha} P'}{P \parallel Q \xrightarrow{\lambda,\textsc{l}\rhd\alpha} P' \parallel Q} \qquad (\textsc{Par-r})\ \frac{Q \xrightarrow{\lambda,\alpha} Q'}{P \parallel Q \xrightarrow{\lambda,\textsc{r}\rhd\alpha} P \parallel Q'}$$

$$(\textsc{Comm})\ \frac{P \xrightarrow{\lambda_1,\alpha_1} P' \quad Q \xrightarrow{\lambda_2,\alpha_2} Q' \quad \gamma(\lambda_1,\lambda_2) = \lambda}{P \parallel Q \xrightarrow{\lambda,\textsc{l}\rhd\alpha_1 \cup \textsc{r}\rhd\alpha_2} P' \parallel Q'}$$

$$(\textsc{Enc})\ \frac{P \xrightarrow{\lambda,\alpha} P' \quad \lambda \notin H}{\partial_H(P) \xrightarrow{\lambda,\alpha} \partial_H(P')}$$

Table 6.1: Structural operational semantics.

Finally, the rule (Enc) expresses that ∂_H blocks transitions labelled with $\lambda \in H$; the set of components is simply inherited.

The example below illustrates the operational rules, and how they can be used to construct derivations.

Example 6.5. *The recursive specification given below models the second situation of Example 6.4, i.e., the situation in which Alice orders coffee and a croissant, and Bob is her waiter.*

$$\mathit{Bob} \overset{\mathit{def}}{=} \mathit{coffee}_r.\mathit{Bob} + \mathit{croissant}_r.\mathit{Bob} + \mathit{phone}.\mathit{Bob} \ \ , \ \ \textit{and}$$

$$\mathit{Alice} \overset{\mathit{def}}{=} \mathit{coffee}_s.\mathit{croissant}_s.\mathbf{0} \ \ ,$$

Assume that γ is a communication function satisfying

$$\gamma(\mathit{coffee}_r, \mathit{coffee}_s) = \mathit{coffee} \ \textit{and}\ \gamma(\mathit{croissant}_r, \mathit{croissant}_s) = \mathit{croissant} \ .$$

Then we can derive the following transition with conclusion $\mathit{Bob} \xrightarrow{\mathit{coffee}_r,\{\epsilon\}} \mathit{Bob}$, with source process Bob, target process Bob, and label coffee_r:

$$(\textsc{Rec})\ \cfrac{(\textsc{Sum-l})\ \cfrac{(\textsc{Pref})\ \cfrac{}{\mathit{coffee}_r.\mathit{Bob} \xrightarrow{\mathit{coffee}_r,\{\epsilon\}} \mathit{Bob}}}{\mathit{coffee}_r.\mathit{Bob} + \mathit{croissant}_r.\mathit{Bob} + \mathit{phone}.\mathit{Bob} \xrightarrow{\mathit{coffee}_r,\{\epsilon\}} \mathit{Bob}}}{\mathit{Bob} \xrightarrow{\mathit{coffee}_r,\{\epsilon\}} \mathit{Bob}}$$

In a similar vein, we can derive a transition that has as conclusion $Alice \xrightarrow{coffee_s,\{\epsilon\}} croissant_s.\mathbf{0}$*, and which allows us to derive a transition witnessing the communication that can take place between Alice and Bob:*

$$(\textsc{Comm})\ \frac{\dfrac{\vdots}{Bob \xrightarrow{coffee_r,\{\epsilon\}} Bob} \qquad \dfrac{\vdots}{Alice \xrightarrow{coffee_s,\{\epsilon\}} croissant_s.\mathbf{0}}}{Bob \parallel Alice \xrightarrow{coffee,\{\mathrm{L},\mathrm{R}\}} Bob \parallel croissant_s.\mathbf{0}}$$

The above derivation shows that both Alice and Bob contribute equally to the transition that results in Alice drinking a cup of coffee.

Now we let Tr be the set of all derivations[1] that can be constructed using the structural operational rules in Table 6.1, and we define src, $target$ and ℓ by stipulating that if $t \in Tr$ is a derivation and $P \xrightarrow{\lambda,\alpha} P'$ is its conclusion, then $src(t) = P$, $target(t) = P'$ and $\ell(t) = \lambda$. Furthermore, we write $comp(t)$ to denote the set of components α contributing to t.

It remains to define the concurrency relation $\smile\!\!\bullet$. We define that transitions t and u are concurrent (notation: $t \smile\!\!\bullet u$) if $comp(t) \cap comp(u) = \emptyset$, i.e., if none of the components contributing to t are contributing to u.

Lemma 6.6. *For all transitions t and v, if $src(t) = src(v)$ and $t \smile\!\!\bullet v$, then there exists a transition u with $src(u) = target(v)$, $\ell(u) = \ell(t)$ and $comp(u) = comp(t)$.*

Proof. By induction on v; for a detailed proof see Lemma 44 in Appendix A of the journal paper [18]. □

Proposition 6.7. *The structure $\mathbf{P} = (St, Tr, src, target, \ell, \smile\!\!\bullet)$ with components as defined above is an LTSC.*

Proof. From the rules in Table 6.1 it is immediate that whenever $P \xrightarrow{\lambda,\alpha} P'$, then $\alpha \neq \emptyset$. So for every $t \in Tr$ we have that $comp(t) \cap comp(t) = \alpha \neq \emptyset$. It follows that $t \not\smile\!\!\bullet t$ and hence $\smile\!\!\bullet$ is irreflexive. That $\smile\!\!\bullet$ also satisfies the second requirement of Definition 6.2 follows with a straightforward induction on the length of π using Lemma 6.6. □

6.3 Modelling Peterson's Algorithm

Peterson's algorithm for mutual exclusion provides a classical solution to enable two processes to use a shared resource in a mutually exclusive manner. In the algorithm, the shared resource is referred to as the *critical section.* The algorithm ensures that at all times only one of the two processes is in the critical section. A desired liveness property of a mutual exclusion algorithm is that whenever one

[1]The notion of derivation with respect to a set of derivation rules can be defined inductively as usual; we omit it here.

of the two processes wishes to enter the critical section, then it will eventually do so. In this section, we shall discuss how Peterson's algorithm can be modelled in the process calculus introduced in the previous section. Then, we shall recap the argument, already presented in [33], that the notion of justness associated with the process calculus is too weak to exclude all unrealistic paths violating the liveness property. In the next section, we shall refine the definition of justness in order to facilitate an exhaustive verification under this notion of justness of the aforementioned liveness property using the mCRL2 toolset.

```
Process A                                      Process B
repeat forever                                 repeat forever
  $\ell_1$ noncritical section                     $m_1$ noncritical section
  $\ell_2$ $readyA := \top$                        $m_2$ $readyB := \top$
  $\ell_3$ $turn := B$                             $m_3$ $turn := A$
  $\ell_4$ await $(readyB = \bot \vee turn = A)$   $m_4$ await $(readyA = \bot \vee turn = B)$
  $\ell_5$ critical section                        $m_5$ critical section
  $\ell_6$ $readyA := \bot$                        $m_6$ $readyB := \bot$
```

Figure 6.1: Peterson's algorithm (pseudocode)

Peterson's algorithm is shown in Figure 6.1. Processes A and B communicate via shared variables. By setting Boolean variables *readyA* and *readyB*, respectively, they signal to the other process their wish to enter the critical section. In addition, a shared variable *turn* is used to keep track of whose turn it is to enter the critical section next; the idea is that a process, before entering its critical section, courteously always first grants access to the other process. This way of using *turn* is essential for ensuring both deadlock freedom and mutual exclusion.

In a message-passing process calculus, global variables are modelled as separate processes with which other processes can interact. Processes modelling a variable keep track of the value of the variable and can communicate with other processes in read and write operations. In our model, to read a variable, the variable that is being read performs an action $s\_rd^{val}_{var}$ and the process that reads the variable performs an action $r\_rd^{val}_{var}$. Together they communicate to a transition labelled with rd^{val}_{var}. A similar communication, labelled with $asgn^{val}_{var}$, is defined to write to a variable. To cover all the interactions with variables in Peterson's algorithm we define the communication function γ_{Pet} in such a way that it satisfies the following equations and is undefined otherwise:

$$\begin{array}{ll}
\gamma_{Pet}(r\_asgn^b_P, s\_asgn^b_P) = asgn^b_P & (b \in \{\top, \bot\}, P \in \{RA, RB\}) \ , \\
\gamma_{Pet}(r\_rd^b_P, s\_rd^b_P) = rd^b_P & (b \in \{\top, \bot\}, P \in \{RA, RB\}) \ , \\
\gamma_{Pet}(r\_asgn^t_T, s\_asgn^t_T) = asgn^t_T & (t \in \{A, B\}) \ , \text{ and} \\
\gamma_{Pet}(r\_rd^t_T, s\_rd^t_T) = rd^t_T & (t \in \{A, B\}) \ .
\end{array} \tag{6.2}$$

We model the behaviour of the three variables *readyA*, *readyB* and *turn* with process identifiers RA^b, RB^b and T^t (with the superscripts referring to the current value of the variable), defined by the following equations:

$$\begin{aligned}
&RA^b = r\_asgn^{\top}_{RA}.RA^{\top} + r\_asgn^{\perp}_{RA}.RA^{\perp} + s\_rd^{b}_{RA}.RA^{b} \quad (b \in \{\top, \perp\}) \ , \\
&RB^b = r\_asgn^{\top}_{RB}.RB^{\top} + r\_asgn^{\perp}_{RB}.RB^{\perp} + s\_rd^{b}_{RB}.RB^{b} \quad (b \in \{\top, \perp\}) \ , \\
&\text{and } T^t = r\_asgn^{A}_{T}.T^{A} + r\_asgn^{B}_{T}.T^{B} + s\_rd^{t}_{T}.T^{t} \quad (t \in \{A, B\}) \ .
\end{aligned}$$

Our specification uses labels **noncritA**, **noncritB**, **critA**, **critB**, to represent exiting the noncritical and critical sections, respectively. Process identifiers *procA* and *procB* model the behaviour of processes A and B. They are defined by the following equations (using the abbreviation $(\lambda_1 + \lambda_2).P$ for $\lambda_1.P + \lambda_2.P$):

$$\begin{aligned}
procA &= \mathbf{noncritA}.s\_asgn^{\top}_{RA}.s\_asgn^{B}_{T}.(r\_rd^{\perp}_{RB} + r\_rd^{A}_{T}). \\
&\quad \mathbf{critA}.s\_asgn^{\perp}_{RA}.procA \\
procB &= \mathbf{noncritB}.s\_asgn^{\top}_{RB}.s\_asgn^{A}_{T}.(r\_rd^{\perp}_{RA} + r\_rd^{B}_{T}). \\
&\quad \mathbf{critB}.s\_asgn^{\perp}_{RB}.procB
\end{aligned}$$

Together, the process definitions form the recursive specification E_{Pet} consisting of eight process identifiers: *procA*, *procB*, $RA^{\top}$, $RA^{\perp}$, $RB^{\top}$, $RB^{\perp}$, T^{A} and T^{B}. With the set H defined by

$$\begin{aligned}
H = \{&s\_asgn^{b}_{P}, r\_asgn^{b}_{P}, s\_rd^{b}_{P}, r\_rd^{b}_{P} \mid b \in \{\top, \perp\}, P \in \{RA, RB\}\} \\
&\cup \{s\_asgn^{t}_{T}, r\_asgn^{t}_{T}, s\_rd^{t}_{P}, r\_rd^{t}_{T} \mid t \in \{A, B\}\} \ ,
\end{aligned}$$

we can now specify Peterson's algorithm with the process expression

$$Pet = \partial_H(procA \parallel (procB \parallel (RA^{\perp} \parallel (RB^{\perp} \parallel T^{A})))) \ .$$

Remark 6.8. *Our specification of Peterson's algorithm is almost identical to the CCS model presented in [33]. The difference is in how communication is defined. CCS presupposes a standard communication function by which an action a can communicate with its co-named action $\bar{a}$, resulting in a special action τ. In our setting, the exact same behaviour as defined by the specification in [33] would be obtained by using, instead of the communication function γ defined above, a communication function γ_{CCS} defined by*

$$\begin{aligned}
&\gamma_{\mathrm{CCS}}(r\_asgn^{b}_{P}, s\_asgn^{b}_{P}) = \tau && (b \in \{\top, \perp\}, P \in \{RA, RB\}) \\
&\gamma_{\mathrm{CCS}}(r\_rd^{b}_{P}, s\_rd^{b}_{P}) = \tau && (b \in \{\top, \perp\}, P \in \{RA, RB\}) \\
&\gamma_{\mathrm{CCS}}(r\_asgn^{t}_{T}, s\_asgn^{t}_{T}) = \tau && (t \in \{A, B\}) \\
&\gamma_{\mathrm{CCS}}(r\_rd^{t}_{T}, s\_rd^{t}_{P}) = \tau && (t \in \{A, B\}) \ .
\end{aligned} \tag{6.3}$$

To get an appropriate notion of just path starting from *Pet*, we define the set of blocking actions

$$\mathcal{B} = \{\mathbf{noncritA}, \mathbf{noncritB}\}$$

Let π denote the unique path starting with *Pet* such that if all states are omitted from it then we obtain the following sequence of labels:

$$\mathbf{noncritA}.(\mathbf{noncritB}.asgn^{\top}_{RB}.asgn^{A}_{T}.rd^{\bot}_{RA}.\mathbf{critB}.asgn^{\bot}_{RB})^{\infty} \enspace .$$

The path π violates the liveness criterion as process A wants to enter the critical section but is never able to, waiting to write to the variable *readyA*. It is deemed unrealistic, as process B reading *readyA* intuitively cannot prevent process A from writing it. To assess whether π is just we need to examine whether every action transition t with $\ell(t) \notin \mathcal{B}$ and $src(t) \in \pi$, a transition u occurs in the suffix of π starting at $src(t)$ such that $t \not\smile^{\bullet} u$. The only component of interest here is *procA* as all other components partake in infinitely many transitions. Let t denote some transition labelled with $asgn^{\top}_{RA}$, with $src(t) \in \pi$. There always exists a transition u labelled with $rd^{\bot}_{RA}$ in the suffix of π starting at $src(t)$. The components partaking in t are L and RRL and the components partaking in u are RL and RRL. Hence, due to the overlap, $t \not\smile^{\bullet} u$; the path violating the liveness property is just.

A more refined definition of the concurrency relation is needed to specify that certain interactions, such as reading a variable, do not interfere with other interactions with the same component. This requires distinguishing between components contributing passively to a transition and components really affected by a transition.

6.4 Signals

In the previous section it was observed that the specification of Peterson's algorithm in the proposed process calculus does not yield the appropriate notion of just path, at least not with the given semantics. The culprit is a combination of two aspects. First, shared variables need to be modelled as separate processes. Second, the process calculus does not offer a facility to distinguish between the activities of reading and writing a variable while, intuitively, if some component reads the value of a variable then this should not prevent another process from writing a new value to it.

The solution proposed in [33] is to extend the syntax of CCS with a *signal emission operator*, in order to treat signals differently in the definition of the concurrency relation. A separate set $\mathcal{S}$ of signals is presupposed, and the signal emission operator adds a λ-labelled self-loop to a state if it can emit signal $\lambda \in \mathcal{S}$. Variables, modelled as processes, then emit their values in the form of signals, and reading the value of a variable can then be treated as not affecting the variable. As a consequence, paths on which some component wants to write to a variable but never succeeds because the variable is perpetually read by some other component are not considered just.

Adding a signal emission operator solves the problem uniformly: with every process expression of the process calculus an appropriate notion of just path is associated: if a component only contributes to a transition by emitting a signal, then this contribution is considered passive. A disadvantage of the solution, however, is that it requires an addition to the syntax of the calculus. As a consequence, standard verification technology such as the mCRL2 toolset, which does not include a signal emission operator, cannot be used to perform verifications taking justness into account.

Here we opt for a different solution, which does not require an addition to the syntax of the process calculus. Instead, it suffices to distinguish a separate set of signals $\mathcal{S}$ and tune the notion of justness to take signals into account. We need to modify the structural operational semantics, giving signals a special status: whenever a transition labelled with a signal indeed does not change state, then it is considered to be a signal. But this modification of the structural operational semantics is only necessary to get an appropriate definition of the concurrency relation. In Sections 6.5 and 6.6, we shall propose sufficient conditions on a process expression (and the underlying recursive specification) that ensure that all transitions labelled with signals are indeed signal transitions. This, in combination with the use of an appropriate communication function that preserves component information, will eventually obviate the need for explicitly defining a concurrency relation on transitions, because it can be deduced from the labelling.

Henceforth we allow $\mathcal{S}$ to be non-empty. The syntax of the process calculus (see Equation (6.1) on p. 121) remains the same. In the structural operational semantics, however, we distinguish between components contributing actively and components contributing passively to a transition. A component contributes passively to a transition if another component reads one of its signals, i.e., the component participates with a transition that is labelled with a signal and this transition does not change the state of the component. The modified structural operational semantics in Table 6.2 defines a transition relation $\xrightarrow{\lambda,\alpha,\varsigma}$ on process expressions, which is endowed with a label $\lambda \in \mathcal{L}$, a set $\alpha \subseteq \mathcal{C}^*$ of *active components* and a set $\varsigma \subseteq \mathcal{C}^*$ of *signalling components.*

Note that $\lambda.P \neq P$, and therefore a transition emanating from a prefix always changes state. Thus, according to the rule (Pref), the transition from a prefix has an active component ϵ and no signalling components.

If an identifier A is the source of a transition that has A also as its target, and this transition is labelled with a signal, then this transition has a signalling component ϵ and no active components; otherwise, the transition has an active component ϵ and no signalling components.

Due to the presence of recursion, it may also happen that $P + Q$ is both the source and the target of a transition, and if such a transition is labelled with a signal, then we want to treat it as a signal transition. This is reflected in (Sum-l) and (Sum-r) by distinguishing whether the target of the transition equals $P + Q$ and is labelled with a signal: if so, then the transition has no active components and a signalling component ϵ; otherwise, the transition has an active component ϵ and no signalling components.

In an application of (Par-l), both the active and signalling components of the premise are prefixed with an l; in an application of (Par-r), they are prefixed with an r; in an application of (Comm) the components of the left premise are

$$(\textsc{Pref})\ \frac{}{\lambda.P \xrightarrow{\lambda,\{\epsilon\},\emptyset} P}$$

$$(\textsc{Rec})\ \frac{P \xrightarrow{\lambda,\alpha,\varsigma} P' \quad A \overset{\text{def}}{=} P}{A \xrightarrow{\lambda,\alpha',\varsigma'} P'} \quad \begin{cases} \alpha' = \emptyset, \varsigma' = \{\epsilon\} & \text{if } \lambda \in \mathcal{S} \text{ and } P' = A \\ \alpha' = \{\epsilon\}, \varsigma' = \emptyset & \text{otherwise} \end{cases}$$

$$(\textsc{Sum-l})\ \frac{P \xrightarrow{\lambda,\alpha,\varsigma} P'}{P + Q \xrightarrow{\lambda,\alpha',\varsigma'} P'} \quad \begin{cases} \alpha' = \emptyset, \varsigma' = \{\epsilon\} & \text{if } \lambda \in \mathcal{S} \text{ and } P' = P + Q \\ \alpha' = \{\epsilon\}, \varsigma' = \emptyset & \text{otherwise} \end{cases}$$

$$(\textsc{Sum-r})\ \frac{Q \xrightarrow{\lambda,\alpha,\varsigma} Q'}{P + Q \xrightarrow{\lambda,\alpha',\varsigma'} Q'} \quad \begin{cases} \alpha' = \emptyset, \varsigma' = \{\epsilon\} & \text{if } \lambda \in \mathcal{S} \text{ and } Q' = P + Q \\ \alpha' = \{\epsilon\}, \varsigma' = \emptyset & \text{otherwise} \end{cases}$$

$$(\textsc{Par-l})\ \frac{P \xrightarrow{\lambda,\alpha,\varsigma} P'}{P \parallel Q \xrightarrow{\lambda,\textsc{l}\rhd\alpha,\textsc{l}\rhd\varsigma} P' \parallel Q} \qquad (\textsc{Par-r})\ \frac{Q \xrightarrow{\lambda,\alpha,\varsigma} Q'}{P \parallel Q \xrightarrow{\lambda,\textsc{r}\rhd\alpha,\textsc{r}\rhd\varsigma} P \parallel Q'}$$

$$(\textsc{Comm})\ \frac{P \xrightarrow{\lambda_1,\alpha_1,\varsigma_1} P' \quad Q \xrightarrow{\lambda_2,\alpha_2,\varsigma_2} Q' \quad \gamma(\lambda_1,\lambda_2) = \lambda}{P \parallel Q \xrightarrow{\lambda,\textsc{l}\rhd\alpha_1\cup\textsc{r}\rhd\alpha_2,\textsc{l}\rhd\varsigma_1\cup\textsc{r}\rhd\varsigma_2} P' \parallel Q'}$$

$$(\textsc{Enc})\ \frac{P \xrightarrow{\lambda,\alpha,\varsigma} P' \quad \lambda \notin H}{\partial_H(P) \xrightarrow{\lambda,\alpha,\varsigma} \partial_H(P')}$$

Table 6.2: Structural operational semantics taking signals into account.

prefixed with an L and those of the right premise are prefixed with an R. In an application of (Enc), both the sets of active and signalling components are simply inherited from the premise.

Example 6.9. *Consider the recursive specification of Peterson's algorithm—and in particular the specification of $RA^\perp$—given in the previous section. Suppose that $s\_rd_{RA}^\perp \in \mathcal{S}$ but $rd_{RA}^\perp, r\_rd_{RA}^\perp \notin \mathcal{S}$. Then we have the following (fragment of a) derivation:*

$$(\textsc{Comm})\ \frac{(\textsc{Rec})\ \dfrac{\vdots}{RA^\perp \xrightarrow{s\_rd_{RA}^\perp,\emptyset,\{\epsilon\}} RA^\perp} \qquad \dfrac{}{r\_rd_{RA}^\perp.\mathbf{0} \xrightarrow{r\_rd_{RA}^\perp,\{\epsilon\},\emptyset} \mathbf{0}}\ (\textsc{Pref})}{RA^\perp \parallel r\_rd_{RA}^\perp.\mathbf{0} \xrightarrow{rd_{RA}^\perp,\{\textsc{r}\},\{\textsc{l}\}} RA^\perp \parallel \mathbf{0}}$$

The component $RA^\perp$ contributes a signal transition, and hence does not actively contribute to the communication. As a consequence, the path we identified earlier as constituting a liveness violation to Peterson's algorithm is, with the revised semantics, no longer just.

We now associate a revised LTSC with our process calculus as follows. Its set of states *St* is again the set of process expressions. Its set of transitions *Tr* is the set of all derivations in accordance with the new structural operational semantics in Table 6.2. Again, if $t \in \mathit{Tr}$ is a derivation with conclusion $P \xrightarrow{\lambda,\alpha,\varsigma} P'$, then $\mathit{src}(t) = P$, $\mathit{target}(t) = P'$ and $\ell(t) = \lambda$. We define the concurrency relation using a refined notion of component, in which we distinguish between *necessary participants* and *affected* components. The set of necessary participants of a transition t, denoted by $\mathit{npc}(t)$, is defined as

$$\mathit{npc}(t) = \alpha \cup \varsigma \ ,$$

and the set of *affected components* of t, denoted by $\mathit{afc}(t)$, is defined as

$$\mathit{afc}(t) = \alpha \ .$$

We define that transitions t and u are concurrent (notation: $t \smile\!\!\!\bullet\, u$) if none of the components necessary for t are affected by u, i.e., if $\mathit{npc}(t) \cap \mathit{afc}(u) = \emptyset$.

To satisfy the requirements on $\smile\!\!\!\bullet$ that it is irreflexive on action transitions, it is important that the set of affected components $\mathit{afc}(t)$ of an action transition t is non-empty, for otherwise $\mathit{npc}(t) \cap \mathit{afc}(t) = \emptyset$. The following example illustrates that we need to formulate some mild restrictions on the communication function for this.

Example 6.10. *Consider the recursive specification consisting of the following two defining equations:*

$$A \overset{def}{=} \lambda_1.A \ , \ \textit{and}$$

$$B \overset{def}{=} \lambda_2.B \ ,$$

and suppose that γ is a communication function satisfying

$$\gamma(\lambda_1,\lambda_2) = \gamma(\lambda_2,\lambda_1) = \lambda_3 \ .$$

Furthermore, suppose that $\lambda_1, \lambda_2 \in \mathcal{S}$, while $\lambda_3 \in \mathcal{A}$. Then we have the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\lambda_1.A \xrightarrow{\lambda_1,\{\epsilon\},\emptyset} A}\,(\textsc{Pref})}{A \xrightarrow{\lambda_1,\emptyset,\{\epsilon\}} A}\,(\textsc{Rec}) \qquad \dfrac{\dfrac{}{\lambda_1.B \xrightarrow{\lambda_2,\{\epsilon\},\emptyset} B}\,(\textsc{Pref})}{B \xrightarrow{\lambda_2,\emptyset,\{\epsilon\}} B}\,(\textsc{Rec})}{A \parallel B \xrightarrow{\lambda_3,\emptyset,\{\textsc{l},\textsc{r}\}} A \parallel B}\,(\textsc{Comm})$$

Since $\lambda_3 \in \mathcal{A}$, this derivation is an action transition, but the set of affected components is empty. The culprit in this example is that communication between the two signals λ_1 and λ_2 results in an action λ_3.

We can exclude the situation as described in the preceding example by requiring that the communication of two signals never results in an action. It is convenient and natural to also require the converse: the communication of an action with another label should never result in a signal.

Definition 6.11. *A communication function γ is* signal-respecting *if and only if for all λ_1 and λ_2, such that $\gamma(\lambda_1, \lambda_2)$ is defined, it is the case that $\gamma(\lambda_1, \lambda_2) \in \mathcal{S}$ if and only if $\lambda_1, \lambda_2 \in \mathcal{S}$.*

Lemma 6.12. *If the communication function γ is signal-respecting, then a transition t is a signal transition if, and only if, $\mathit{afc}(t) = \emptyset$.*

Proof. By induction on t; for a detailed proof see Lemma 45 in Appendix A of the journal paper [18]. □

In the following corollary, which is an immediate consequence of the preceding lemma, we establish that $\smile^{\bullet}$ satisfies condition 1 of Definition 6.2.

Corollary 6.13. *If the communication function γ is signal-respecting, then $\smile^{\bullet}$ is irreflexive on action transitions, i.e., for all action transitions t we have $t \not\smile^{\bullet} t$.*

Proof. Let t be an action transition. Then, by Lemma 6.12, $\mathit{afc}(t) \neq \emptyset$. Since $\mathit{afc}(t) \subseteq \mathit{npc}(t)$, it follows that $\mathit{npc}(t) \cap \mathit{afc}(t) \neq \emptyset$, and hence $t \not\smile^{\bullet} t$. □

Lemma 6.14. *For all transitions t and v, if $\mathit{src}(t) = \mathit{src}(v)$ and $\mathit{npc}(t) \cap \mathit{afc}(v) = \emptyset$, then there exists a transition u with $\mathit{src}(u) = \mathit{target}(v)$, $\ell(u) = \ell(t)$ and $\mathit{npc}(u) = \mathit{npc}(t)$. If γ is signal-respecting and t is an action transition, then so is u.*

Proof. By induction on v; for a detailed proof see Lemma 46 in Appendix A of the journal paper [18]. □

It follows from the preceding lemma that the relation $\smile^{\bullet}$ associated with our process calculus satisfies condition 2 of Definition 6.2, as established in the following corollary.

Corollary 6.15. *If γ is signal-respecting, t is an action transition and π is a path from $\mathit{src}(t)$ to some process expression P such that $t \smile^{\bullet} v$ for all transitions v occurring on π, then there is an action transition u such that $\mathit{src}(u) = P$, $\ell(u) = \ell(t)$ and $t \not\smile^{\bullet} u$.*

Proof. Straightforward induction on the length of π using Lemma 6.14. □

From Corollaries 6.13 and 6.15 we get the following proposition.

Proposition 6.16. *Let γ be signal-respecting, let $\mathit{St} = \mathcal{P}$, let Tr be the set of all derivations of transitions in accordance with the operational semantics, stipulating that if $t \in \mathit{Tr}$ is a derivation with conclusion $P \xrightarrow{\lambda,\alpha,\varsigma} P'$, then $\mathit{src}(t) = P$, $\mathit{target}(t) = P'$ and $\ell(t) = \lambda$, $\mathit{npc}(t) = \alpha \cup \varsigma$ and $\mathit{afc}(t) = \alpha$, and defining $\smile^{\bullet}$ by $t \smile^{\bullet} u$ if, and only if, $\mathit{npc}(t) \cap \mathit{afc}(u) = \emptyset$. Then*

$$\mathbf{P} = (\mathit{St}, \mathit{Tr}, \mathit{src}, \mathit{target}, \ell, \smile^{\bullet})$$

is an LTSC.

Example 6.17. *Returning to the running example of Peterson's algorithm we reconsider the path that violates liveness. First, we define the signal actions and check whether the communication function is signal-respecting.*

$$\mathcal{S} = \{s\_rd^{b}_{P} \mid b \in \{\top, \bot\}, P \in \{RA, RB\}\} \cup \{s\_rd^{t}_{T} \mid t \in \{A, B\}\}$$

It is easy to see that the communication function γ, defined in Equation (6.2) *on p. 125, is signal-respecting. Taking the LTSC as defined in Proposition 6.16 we re-examine the liveness violating path π presented at the end of Section 6.3, which gives rise to the following sequence of labels:*

$$\textbf{noncritA}.(\textbf{noncritB}.asgn^{\top}_{RB}.asgn^{A}_{T}.rd^{\bot}_{RA}.\textbf{critB}.asgn^{\bot}_{RB})^{\infty}.$$

Let t and u be any two transitions with labels $asgn^{\top}_{RA}$ and $rd^{\bot}_{RA}$, respectively. Then $npc(t) = \{\text{L}, \text{RRL}\}$ and $afc(u) = \{\text{RL}\}$. Therefore $npc(t) \cap afc(u) = \emptyset$ and thus $t \smile^{\bullet} u$. We conclude that path π contains transition t, $src(t) \in \pi$, for which there does not exist a transition v in the suffix of π such that $t \not\smile^{\bullet} v$. The path π is therefore not just and can be ruled out. Note that this does not constitute a proof of liveness, we have only reasoned about a single path. To prove liveness we need to prove that there does not exist another liveness violating path that is just.

6.5 Concurrency-consistent Labelling

The semantics we associated with our process calculus in the previous section enables reasoning about just paths without the need for additional operators in the language. This allows one to manually analyse, e.g., the required liveness property of Peterson's algorithm in a standard process algebra, by reasoning directly about the relevant just paths in the LTSC under analysis. Our aim, however, is to facilitate the automated verification of liveness properties for just paths, using toolsets such as mCRL2. Such toolsets are based on labelled transition systems without a concurrency relation. Moreover, in these toolsets, properties need to be expressed in a modal logic that has modalities that refer to labels, and not to individual transitions.

Our specification of Peterson's algorithm is such that it allows a characterisation of its just paths in terms of labels rather than referring to individual transitions in the LTSC. This is possible, because the labelling of transitions reachable from *Pet* is consistent with the concurrency relation on those transitions.

In this section, we formally define when an LTSC has a concurrency-consistent labelling, and we prove that LTSCs with a concurrency-consistent labelling allow a characterisation of just paths in terms of labels instead of individual transitions. In the next section, we shall provide a sufficient syntactic criterion on specifications in our process calculus that ensure that the associated LTSC has a concurrency-consistent labelling, and we argue that our specification of Peterson's algorithm satisfies this syntactic criterion.

Definition 6.18. *An LTSC* $(St, Tr, src, target, \ell, \smile^{\bullet})$ *has a* concurrency-consistent labelling *if for every* $t \in Tr$, $\ell(t) \in \mathcal{S}$ *implies* $src(t) = target(t)$, *and there exists a binary relation* $\smile^{\bullet}$ *on the set of labels* $\mathcal{L}$ *such that for all transitions* $t, u \in Tr$ *we have that* $t \smile^{\bullet} u$ *if, and only if,* $\ell(t) \smile^{\bullet} \ell(u)$.

Clearly, there is no harm in the overloading of the symbol $\smile^{\bullet}$. In an LTSC with a concurrency-consistent labelling the relation on $\mathcal{L}$ is uniquely determined by the relation on Tr. Furthermore, it will be clear from the context whether we mean the relation on transitions or the relation on labels. For an LTSC with concurrency-consistent labelling, we can reformulate the notion of $\mathcal{B}$-justness referring to labels instead of transitions. A label $\lambda \in \mathcal{L}$ is *enabled* in a state $s \in St$ if there is a transition t with $src(t) = s$ and $\ell(t) = \lambda$. An action $\lambda \in \mathcal{A}$ is *eliminated* on a path π if there is a transition t on π such that $\lambda \not\smile^{\bullet} \ell(t)$. In an LTSC with a concurrency-consistent labelling, action transitions are not labelled by signals, so a non-blocking action transition is labelled by an element of the complement $\overline{\mathcal{B}} = \mathcal{A} \backslash \mathcal{B}$ of $\mathcal{B}$ relative to $\mathcal{A}$.

Proposition 6.19. *Let* $(St, Tr, src, target, \ell, \smile^{\bullet})$ *be an LTSC with a concurrency-consistent labelling. Let* $\mathcal{B} \subseteq \mathcal{A}$ *be a set of blocking actions. A path* π *is then* $\mathcal{B}$*-just if, and only if, for every state* s *on* π *and every* $\lambda \in \overline{\mathcal{B}}$ *enabled in* s, λ *is eliminated in the suffix of* π *starting at* s.

Proof. Let π be a path in $(St, Tr, src, target, \ell, \smile^{\bullet})$.

To prove the implication from left to right, suppose that π is $\mathcal{B}$-just and suppose that $\lambda \in \overline{\mathcal{B}}$ is enabled in some state s on π. Then there is an action transition t with $src(t) = s$ and $\ell(t) = \lambda$, so, by $\mathcal{B}$-justness, a transition u occurs in the suffix of π starting at $src(t) = s$ such that $t \not\smile^{\bullet} u$. Since the LTSC has a concurrency-consistent labelling, it follows that $\lambda = \ell(t) \not\smile^{\bullet} \ell(u)$, and hence λ is eliminated on the suffix of π starting at s.

To prove the implication from right to left, let t be an action transition such that $\ell(t) \notin \mathcal{B}$ and $src(t) \in \pi$. Then $\ell(t)$ is enabled and, since t is an action transition and the LTSC has a concurrency-consistent labelling, it follows that $\ell(t) \in \overline{\mathcal{B}}$, so λ is eliminated in the suffix of π starting at $src(t)$. So there is a transition u in the suffix of π starting at $src(t)$ such that $\ell(t) \not\smile^{\bullet} \ell(u)$. Hence, since the LTSC has a concurrency-consistent labelling, $t \not\smile^{\bullet} u$, confirming that π is $\mathcal{B}$-just. □

6.6 Syntactic Conditions

The LTSC **P** associated with the process calculus in Section 6.4 does not have a concurrency-consistent labelling, simply because there exist process expressions (e.g., $\lambda.\mathbf{0}$ with $\lambda \in \mathcal{S}$) that give rise to state-changing transitions labelled with signals. In automated verification, however, we are often only interested in the restriction of **P** to the set of process expressions reachable from some initial process expression; for example, when verifying Peterson's algorithm we are only interested in states and transitions reachable from *Pet*. We shall now first formally define the LTSC associated with a process expression P and then formulate sufficient syntactic conditions that guarantee that this LTSC has a concurrency-consistent labelling, all in the context of a specific communication function.

Definition 6.20. *Let P be a process expression. The* LTSC associated with P *has as set of states the set of all process expressions reachable from P in* $\mathbf{P}$*, as transitions the set of all transitions reachable from P, and functions src, target, ℓ and relation $\smile\!\!\bullet$ obtained by restricting those of* $\mathbf{P}$ *to the set of transitions reachable from P.*

In Section 6.4, the concurrency relation $\smile\!\!\bullet$ on transitions was derived from assignments $\mathit{npc} : \mathit{Tr} \to 2^{\mathcal{C}^*}$ and $\mathit{afc} : \mathit{Tr} \to 2^{\mathcal{C}^*}$ of necessary participants and affected components to individual transitions. It is convenient to formulate sufficient conditions in terms of assignments $\mathit{npc}_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ and $\mathit{afc}_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ of necessary and affected components to labels, respectively, satisfying for every transition t

$$\mathit{npc}(t) = \mathit{npc}_\ell(\ell(t)), \text{ and} \tag{6.4}$$

$$\mathit{afc}(t) = \mathit{afc}_\ell(\ell(t)) \ . \tag{6.5}$$

It is not possible to satisfy these equations in general: an appropriate assignment of components to labels largely depends on the process expression under consideration. Moreover, it may not even be possible to define $\mathit{npc}_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ and $\mathit{afc}_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ in such a way that the equations above are satisfied for all reachable transitions.

Example 6.21. *Consider the specification Pet of Peterson's algorithm presented in Section 6.3, and consider the state reached from Pet by first executing* **noncritA** *and then executing* **noncritB***. In that state, two transitions are enabled: let us denote by t the transition corresponding to the activity of process A assigning the value $\top$ to the variable readyA (this is statement ℓ_2 in Figure 6.1) and let us denote by u the transition corresponding to the activity of process B assigning the value $\top$ to the variable readyB (this is statement m_2 in Figure 6.1). Then $\mathit{npc}(t) = \{\mathrm{L}, \mathrm{RRL}\}$ and $\mathit{npc}(u) = \{\mathrm{RL}, \mathrm{RRRL}\}$. Now observe that, in the context of the CCS communication function γ_{CCS}, defined in Equation* (6.3) *on p. 126, we have that $\ell(t) = \ell(u) = \tau$, and hence it is not possible to define a mapping $\mathit{npc}_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ satisfying Equation* (6.4)*. Note that with the communication function γ, defined in Equation* (6.2) *on p. 125 the problem disappears, since t and u have distinct labels $\mathit{asgn}_{RA}^{\top}$ and $\mathit{asgn}_{RB}^{\top}$, respectively.*

The goal in this section is to formulate sufficient conditions on the communication function γ and a process expression P that allow us to define npc_ℓ and afc_ℓ satisfying Equations (6.4) and (6.5) for all transitions t reachable from P. Furthermore, we show that our specification of Peterson's algorithm satisfies these restrictions.

We first formulate some basic requirements on npc_ℓ and afc_ℓ, expressing that the set of affected components associated with a label is included in the set of necessary components, and that signals do not have active components.

Definition 6.22. *Let $C \subseteq \mathcal{C}^*$ be a finite set of static components. A C-assignment is a pair $(\mathit{npc}_\ell, \mathit{afc}_\ell)$ of mappings $\mathit{npc}_\ell, \mathit{afc}_\ell : \mathcal{L} \to 2^C$ such that*

1. *$afc_\ell(\lambda) \subseteq npc_\ell(\lambda)$ for all $\lambda \in \mathcal{L}$; and*
2. *$afc_\ell(\lambda) = \emptyset$ for all $\lambda \in \mathcal{S}$.*

In the following example, we define a $\mathcal{C}(Pet)$-assignment for our specification of Peterson's algorithm.

Example 6.23. *Recall that the set of components $\mathcal{C}(Pet)$ associated with Pet is*

$$\mathcal{C}(Pet) = \{\text{L},\ \text{RL},\ \text{RRL},\ \text{RRRL},\ \text{RRRR}\}\ .$$

To define the mappings $npc_\ell, afc_\ell : \mathcal{L} \to 2^C$ it is convenient to first associate with every component $\sigma \in \mathcal{C}(Pet)$ a set of labels $\mathcal{L}_\sigma \subseteq \mathcal{L}$. We have

$$\begin{aligned}
\mathcal{L}_{\text{L}} &= \{\textbf{noncritA}, s\_\,asgn_{RA}^{\top}, s\_\,asgn_{T}^{B}, r\_\,rd_{RB}^{\perp}, r\_\,rd_{T}^{A},\\
&\qquad \textbf{critA}, s\_\,asgn_{RA}^{\perp}\}\ ,\\
\mathcal{L}_{\text{RL}} &= \{\textbf{noncritB}, s\_\,asgn_{RB}^{\top}, s\_\,asgn_{T}^{A}, r\_\,rd_{RA}^{\perp}, r\_\,rd_{T}^{B},\\
&\qquad \textbf{critB}, s\_\,asgn_{RB}^{\perp}\}\ ,\\
\mathcal{L}_{\text{RRL}} &= \{r\_\,asgn_{RA}^{\top}, r\_\,asgn_{RA}^{\perp}, r\_\,rd_{RA}^{\top}, r\_\,rd_{RA}^{\perp}\}\ ,\\
\mathcal{L}_{\text{RRRL}} &= \{r\_\,asgn_{RB}^{\top}, r\_\,asgn_{RB}^{\perp}, r\_\,rd_{RB}^{\top}, r\_\,rd_{RB}^{\perp}\}\ ,\ \textit{and}\\
\mathcal{L}_{\text{RRRR}} &= \{r\_\,asgn_{T}^{A}, r\_\,asgn_{T}^{B}, r\_\,rd_{T}^{A}, r\_\,rd_{T}^{B}\}\ .
\end{aligned}$$

Now we can define, for all $\sigma \in \mathcal{C}(Pet)$ and all $\lambda \in \mathcal{L}_\sigma$:

$$npc_\ell(\lambda) = \{\sigma\},\ \textit{and}\ afc_\ell(\lambda) = \begin{cases} \{\sigma\} & \textit{if}\ \lambda \in \mathcal{A},\ \textit{and}\\ \emptyset & \textit{if}\ \lambda \in \mathcal{S}\ . \end{cases}$$

On the other elements of $\mathcal{L}$, the results of communications, npc_ℓ and afc_ℓ are defined as follows:

$$\begin{array}{ll}
npc_\ell(asgn_{RA}^{b}) = afc_\ell(asgn_{RA}^{b}) = \{\text{L}, \text{RRL}\} & (b \in \{\top, \perp\})\ ,\\
npc_\ell(asgn_{RB}^{b}) = afc_\ell(asgn_{RB}^{b}) = \{\text{RL}, \text{RRRL}\} & (b \in \{\top, \perp\})\ ,\\
npc_\ell(asgn_{T}^{B}) = afc_\ell(asgn_{T}^{B}) = \{\text{L}, \text{RRRR}\}\ , & \\
npc_\ell(asgn_{T}^{A}) = afc_\ell(asgn_{T}^{A}) = \{\text{RL}, \text{RRRR}\}\ , & \\
npc_\ell(rd_{RA}^{b}) = \{\text{RL}, \text{RRL}\},\quad afc_\ell(rd_{RA}^{b}) = \{\text{RL}\} & (b \in \{\top, \perp\})\ ,\\
npc_\ell(rd_{RB}^{b}) = \{\text{L}, \text{RRRL}\},\quad afc_\ell(rd_{RB}^{b}) = \{\text{L}\} & (b \in \{\top, \perp\})\ ,\\
npc_\ell(rd_{T}^{A}) = \{\text{L}, \text{RRRR}\},\quad afc_\ell(rd_{T}^{A}) = \{\text{L}\}\ ,\ \textit{and} & \\
npc_\ell(rd_{T}^{B}) = \{\text{RL}, \text{RRRR}\},\quad afc_\ell(rd_{T}^{B}) = \{\text{RL}\}\ . &
\end{array}$$

It is easy to verify that (npc_ℓ, afc_ℓ) satisfies the requirements of Definition 6.22 and hence is a $\mathcal{C}(Pet)$-assignment.

We could now proceed to prove directly that the $\mathcal{C}(Pet)$-assignment in the preceding example satisfies Equations (6.4) and (6.5) and conclude that the LTSC associated with *Pet* has a concurrency-consistent labelling. We prefer to proceed more generally, however, and define a subclass of process expressions together with assumptions on the underlying recursive specification that guarantee that an assignment satisfying Equations (6.4) and (6.5) exists. It will be easy to verify that *Pet* is a process expression in the subclass, and that the recursive specification E_{Pet} satisfies the assumptions, from which it will follow that the $\mathcal{C}(Pet)$-assignment above indeed satisfies Equations (6.4) and (6.5). In fact, it can be checked automatically whether a process expression is in the subclass and the underlying recursive specification satisfies the assumptions.

We consider parallel compositions of sequential components. These sequential components should have disjoint alphabets and respect the use of signals. Moreover, the communication function should support a consistent assignment of components to labels. Below, we shall first formulate sufficient conditions on a sequential process expression and its underlying sequential recursive specification that ensure that transitions labelled with signals do not change state in the LTSC associated with the process expression. Then, we associate with every sequential process expression its (reachable) alphabet and its (reachable) action alphabet, so that we can formulate the requirement that the alphabets of components are disjoint. And finally we shall define when an assignment is consistent with a communication function.

Sequential Components. The set of *sequential process expressions* is generated by the following grammar (with A ranging over $\mathcal{I}$ and λ ranging over $\mathcal{L}$):

$$S ::= \mathbf{0} \mid \lambda.S \mid S+S \mid A \ .$$

By a *sequential recursive specification* E we mean a set of defining equations

$$A \stackrel{\text{def}}{=} S_A \ ,$$

with S_A a sequential process expression, including precisely one such equation for every $A \in \mathcal{I}$.

A sequential process expression S is *syntactically guarded* if all occurrences of process identifiers in S are within the scope of an action prefix. A sequential recursive specification E is *syntactically guarded* if for every defining equation $A \stackrel{\text{def}}{=} S_A$ in E it holds that S_A is syntactically guarded.

Respect for Signals. Let E be a sequential recursive specification, and let us denote, for all $A \in \mathcal{I}$, by S_A the right-hand side of the defining equation for A in E. We say that $A \in \mathcal{I}$ is *signalling* if S_A has a subexpression $\lambda.A$ with $\lambda \in \mathcal{S}$. A process identifier $A \in \mathcal{I}$ is *signal-respecting* if

1. for every subexpression $\lambda.S'$ of S_A with $\lambda \in \mathcal{S}$ it holds that $S' = A$ and the occurrence of the subexpression is not in the scope of another prefix, and
2. for every subexpression $S_1 + S_2$ of S_A it holds that S_1 and S_2 are not signalling process identifiers.

E is *signal-respecting* if it is syntactically guarded and all process identifiers in $\mathcal{I}$ are signal-respecting. A sequential process expression S is *signal-respecting* with respect to a signal-respecting sequential recursive specification E if S does not have subexpressions of the form $\lambda.S'$ with $\lambda \in \mathcal{S}$, and for every subexpression $S_1 + S_2$ it holds that S_1 and S_2 are not signalling process identifiers.

Example 6.24. *It is straightforward to check that E_{Pet} is a syntactically guarded sequential recursive specification and that it is signal-respecting.*

Lemma 6.25. *Let E be a signal-respecting recursive specification and let t be a transition such that $src(t)$ is a signal-respecting sequential process expression. Then $target(t)$ is again a signal-respecting sequential process expression, and t is a signal transition if, and only if, $\ell(t) \in \mathcal{S}$.*

Proof. To establish that $target(t)$ is again a signal-respecting sequential process expression, we first note that if $A \overset{\text{def}}{=} S_A$ is the equation in E defining some process identifier A, and $S_A \xrightarrow{\lambda,\alpha,\varsigma} S'$, then S' is signal-respecting. For by syntactic guardedness, S' is a subexpression of S_A, by the first requirement satisfied by signal-respecting process identifiers S' cannot have subexpressions of the form $\lambda.S''$ with $\lambda \in \mathcal{S}$, and by the second requirement satisfied by signal-respecting process identifiers, whenever $S_1 + S_2$ is a subexpression of S', then S_1 and S_2 cannot be signalling process identifiers. We can now argue that $target(t)$ is a signal-respecting sequential process expression with a straightforward induction on the structure of $src(t)$.

It remains to show that t is a signal transition if, and only if, $\ell(t) \in \mathcal{S}$.

For the implication from left to right, note that if t is a signal transition, then, by definition, $\ell(t) \in \mathcal{S}$.

For the converse implication, suppose that $\ell(t) \in \mathcal{S}$; we need to establish that $src(t) = target(t)$. To this end, we first establish with induction on the structure of S that if S is a signal-respecting process expression, $\lambda \in \mathcal{S}$ and $S \xrightarrow{\lambda,\alpha,\varsigma} S'$, then $S = A$ for some process identifier A. Clearly, S cannot be $\mathbf{0}$. Furthermore, since signal-respecting sequential process expressions do not have subexpressions of the form $\lambda.S''$ with $\lambda \in \mathcal{S}$, we cannot have that $S = \lambda.S''$ for some process expression S''. Note that if we had $S = S_1 + S_2$, then either $S_1 \xrightarrow{\lambda,\alpha,\varsigma} S'$ or $S_2 \xrightarrow{\lambda,\alpha,\varsigma} S'$, so by the induction hypothesis either S_1 or S_2 would be a signalling process identifier, contradicting the assumption that for every subexpression $S_1 + S_2$ of S it holds that S_1 and S_2 are not signalling process identifier. It follows that $S = A$ for some (signalling) process identifier A. Hence, assuming that $(A \overset{\text{def}}{=} S_A) \in E$, t has a subderivation t' with $src(t') = S_A$ and $\ell(t') \in \mathcal{S}$. From the first requirement satisfied by signal-respecting process identifiers it now follows that $target(t) = target(t') = A$. □

Alphabet. We also wish to associate with each sequential process expression S its *alphabet* $\mathcal{L}(S)$ and its *action alphabet* $\mathcal{A}(S)$, the sets of labels of transitions and action transitions reachable from S, respectively. To this end, we first define

$\mathcal{L}(A)$ for all process identifiers defined in E, using two auxiliary notions. First, we associate with every sequential process expression S its non-recursive alphabet $\mathcal{L}'(S)$ inductively by: $\mathcal{L}'(\mathbf{0}) = \emptyset$, $\mathcal{L}'(A) = \emptyset$ for all $A \in \mathcal{I}$, $\mathcal{L}'(\lambda.S) = \{\lambda\} \cup \mathcal{L}'(S)$, and $\mathcal{L}'(S_1 + S_s) = \mathcal{L}'(S_1) \cup \mathcal{L}'(S_2)$. Second, we define on $\mathcal{I}$ a binary relation $\rhd$ by $A \rhd A'$ if $A \stackrel{\text{def}}{=} S$ in E and A' occurs in S, and denote by $\rhd^*$ the reflexive-transitive closure of $\rhd$. Then we can define the alphabet $\mathcal{L}(A)$ of A by

$$\mathcal{L}(A) = \bigcup\{\mathcal{L}'(S) \mid A \rhd^* A' \text{ and } A' \stackrel{\text{def}}{=} S\} \ .$$

Now, we inductively extend $\mathcal{L}(\_)$ to all sequential process expressions defining $\mathcal{L}(\mathbf{0}) = \emptyset$, $\mathcal{L}(\lambda.S) = \{\lambda\} \cup \mathcal{L}(S)$, and $\mathcal{L}(S_1 + S_2) = \mathcal{L}(S_1) \cup \mathcal{L}(S_2)$. Furthermore, we define $\mathcal{A}(S) = \mathcal{L}(S) \cap \mathcal{A}$.

Lemma 6.26. *Let E be a sequential recursive specification and let S be a sequential process expression over E. If S' is a sequential process expression reachable from S, then $\mathcal{L}(S') \subseteq \mathcal{L}(S)$ and $\mathcal{A}(S') \subseteq \mathcal{A}(S)$.*

Proof. We first consider the special case that there is a transition t with $\mathit{src}(t) = S$ and $\mathit{target}(t) = S'$ and prove with induction on t that $\mathcal{L}(S') \subseteq \mathcal{L}(S)$.

If the last rule applied in t is (Pref), then we have $S = \lambda.S'$ and hence $\mathcal{L}(S') \subseteq \{\lambda\} \cup \mathcal{L}(S') = \mathcal{L}(S)$.

If the last rule applied in t is (Sum-l), then there exist S_1 and S_2 such that $S = S_1 + S_2$, and t has a subderivation t' with $\mathit{src}(t') = S_1$ and $\mathit{target}(t') = S'$. By the induction hypothesis we have that $\mathcal{L}(S') \subseteq \mathcal{L}(S_1) \subseteq \mathcal{L}(S_1) \cup \mathcal{L}(S_2) = \mathcal{L}(S)$.

If the last rule applied in t is (Sum-r), then there exist S_1 and S_2 such that $S = S_1 + S_2$, and t has a subderivation t' with $\mathit{src}(t') = S_2$ and $\mathit{target}(t') = S'$. By the induction hypothesis we have that $\mathcal{L}(S') \subseteq \mathcal{L}(S_2) \subseteq \mathcal{L}(S_1) \cup \mathcal{L}(S_2) = \mathcal{L}(S)$.

If the last rule applied in t is (Rec), then $S = A$ for some process identifier $A \in \mathcal{I}$ with defining equation $(A \stackrel{\text{def}}{=} S_A) \in E$, and t has a subderivation t' with $\mathit{src}(t') = S_A$ and $\mathit{target}(t') = S'$. By the induction hypothesis, $\mathcal{L}(S') \subseteq \mathcal{L}(S_A)$; it therefore remains to show that $\mathcal{L}(S_A) \subseteq \mathcal{L}(A)$. We have:

$$\begin{aligned}
\mathcal{L}(S_A) &= \mathcal{L}'(S_A) \cup \bigcup\{\mathcal{L}(A') \mid A \rhd A'\} \\
&= \mathcal{L}'(S_A) \cup \bigcup\{\mathcal{L}'(S_{A''}) \mid A \rhd A' \rhd^* A''\} \\
&= \bigcup\{\mathcal{L}'(S_{A'}) \mid A \rhd^* A'\} \\
&= \mathcal{L}(A) \ .
\end{aligned}$$

(In the second equality we have used that $A \rhd^* A''$ for all A' such that $A \rhd A'$. In the third equality we have used the definition of $\mathcal{L}(A)$.)

Now, if S' is reachable from S, then the statement of the lemma follows with a straightforward induction on the number of transitions in a path from S to S'. Furthermore, it is then immediate from the definition of action alphabet that $\mathcal{A}(S') \subseteq \mathcal{A}(S)$. □

Parallel-sequential Processes. Presupposing a signal-respecting sequential recursive specification E, a *parallel-sequential* process expression over E is a process expression generated by the following grammar (with S ranging over sequential process expressions and $H \subseteq \mathcal{L}$):

$$P ::= S \mid P \parallel P \mid \partial_H(P) \ .$$

Lemma 6.27. *Let E be a sequential recursive specification and let P be a parallel-sequential process expression over E. If P' is reachable from P, then $\mathcal{C}(P') = \mathcal{C}(P)$ and $P'|_\sigma$ is reachable from $P|_\sigma$ for all $\sigma \in \mathcal{C}(P)$.*

Proof. With induction on t it can be established that if t is a transition such that $src(t) = P$ and $target(t) = P'$, then $\mathcal{C}(P') = \mathcal{C}(P)$ and $P'|_\sigma = P|_\sigma$ for all $\sigma \in \mathcal{C}(P)$. The details are worked out in the proof of Lemma 47 in Appendix B of the journal paper [18].

Then, if P' is reachable from P, the statement of the lemma follows with a straightforward induction on the number of transitions in a path from P to P'. □

Since a communication function γ is required to be commutative and associative, it induces a partial function $\overline{\gamma} : \mathcal{M}_f(\mathcal{L}) \rightharpoonup \mathcal{L}$, where $\mathcal{M}_f(\mathcal{L})$ denotes the set of all finite multisets over $\mathcal{L}$. We define $\overline{\gamma}([\lambda_0, \dots, \lambda_n])$ with induction on n as follows:

1. If $n = 0$, then $\overline{\gamma}([\lambda_0, \dots, \lambda_n]) = \lambda_0$.
2. If $n = 1$, then $\overline{\gamma}([\lambda_0, \dots, \lambda_n]) = \gamma(\lambda_0, \lambda_n)$ if $\gamma(\lambda_0, \lambda_n)$ is defined, and undefined otherwise.
3. If $n \geq 1$, then $\overline{\gamma}([\lambda_0, \dots, \lambda_{n+1}]) = \gamma(\overline{\gamma}([\lambda_0, \dots, \lambda_n]), \lambda_{n+1})$ in the case that both $\overline{\gamma}([\lambda_0, \dots, \lambda_n])$ and $\gamma(\overline{\gamma}([\lambda_0, \dots, \lambda_n]), \lambda_{n+1})$ are defined, and undefined otherwise.

It is straightforward to prove, with induction on $n \geq 1$, that for all $\lambda_0, \dots, \lambda_n$ and for all $0 \leq k < n$ that $\gamma(\overline{\gamma}(\lambda_0, \dots, \lambda_k), \overline{\gamma}(\lambda_{k+1}, \dots, \lambda_n)) = \overline{\gamma}(\lambda_0, \dots, \lambda_n)$; we shall use this fact in the proof of the next lemma, which relates transitions of a parallel-sequential process with transitions of its components.

Lemma 6.28. *Let t be a transition, let $npc(t) = \{\sigma_0, \dots, \sigma_n\}$, and suppose that $src(t)$ is a parallel-sequential process expression. Then t has subderivations $t_0, \dots, t_n$ such that $src(t_i)$ is a sequential process expression and $src(t_i) = src(t)|_{\sigma_i}$ for all $0 \leq i \leq n$, and $\ell(t) = \overline{\gamma}([\ell(t_0), \dots, \ell(t_n)])$ (where $[\ell(t_0), \dots, \ell(t_n)]$ denotes the multiset over $\mathcal{L}$ consisting of $\ell(t_0), \dots, \ell(t_n)$).*

Proof. We proceed by induction on t.

If the last rule applied in t is (Pref), (Sum-l), (Sum-r), or (Rec), then $npc(t) = \{\epsilon\}$, $src(t) = src(t)|_\epsilon$, and $\ell(t) = \overline{\gamma}([\ell(t)])$. Moreover, from the syntax definition of parallel-sequential processes it is clear that $src(t)$ is a sequential process expression.

If the last rule applied in t is (Par-l), then t has a subderivation t' such that $npc(t) = \mathrm{L} \rhd npc(t')$, so there exist static components $\sigma'_0, \dots, \sigma'_n$ such that $\sigma_i = \mathrm{L}\sigma'_i$

for all $0 \leq i \leq n$. By the induction hypothesis, t', and hence t, has subderivations $t_0, \ldots, t_n$ such that $src(t_i)$ is a sequential process expression, $src(t_i) = src(t')|_{\sigma'_i} = src(t)|_{\sigma}$ for all $0 \leq i \leq n$ and $\ell(t) = \ell(t') = \overline{\gamma}([\ell(t_0), \ldots, \ell(t_n)])$.

If the last rule applied in t is (Par-r), then the proof proceeds analogously.

If the last rule applied in t is (Comm), then t has subderivations t' and t'' such that $npc(t) = \text{L} \rhd npc(t') \cup \text{R} \rhd npc(t'')$. Since $npc(t')$ and $npc(t'')$ cannot be empty, we have that $n \geq 1$ and there exist static components $\sigma'_0, \ldots, \sigma'_n$ and a $0 \leq k < n$ such that $npc(t') = \{\sigma'_0, \ldots, \sigma'_k\}$ and $npc(t'') = \{\sigma'_{k+1}, \ldots, \sigma_n\}$. By the induction hypothesis, t' and t'', and hence t, have subderivations $t_0, \ldots, t_n$ such that $src(t_i)$ is a sequential process expression for all $0 \leq i \leq n$, $src(t_i) = src(t')|_{\sigma'_i} = src(t)|_{\sigma_i}$ for all $0 \leq i \leq k$, $\ell(t') = \overline{\gamma}([\ell(t_0), \ldots, \ell(t_k)])$, $src(t_i) = src(t'')|_{\sigma'_i} = src(t)|_{\sigma_i}$ for all $k < i \leq n$, and $\ell(t'') = \overline{\gamma}([\ell(t_{k+1}), \ldots, \ell(t_n)])$. Furthermore, we have that

$$\begin{aligned}\ell(t) = \gamma(\ell(t'), \ell(t'')) &= \gamma(\overline{\gamma}([\ell(t_0), \ldots, \ell(t_k)]), \overline{\gamma}([\ell(t_{k+1}), \ldots, \ell(t_n)])) \\ &= \overline{\gamma}([\ell(t_0), \ldots, \ell(t_n)]) \ .\end{aligned}$$

Finally, if the last rule applied in t is (Enc), then t has a subderivation t' with $npc(t') = \{\sigma_0, \ldots, \sigma_n\}$ and $\ell(t') = \ell(t)$, so it follows immediately by the induction hypothesis that there exist subderivations $t_0, \ldots, t_n$ of t' and hence of t such that $src(t_i) = src(t')|_{\sigma_i} = src(t)|_{\sigma_i}$ and $\ell(t) = \ell(t') = \overline{\gamma}([\ell(t_0), \ldots, \ell(t_n)])$. □

Definition 6.29. *Let $C \subseteq \mathcal{C}^*$ be a finite set of static components. A C-assignment (npc_ℓ, afc_ℓ) is* consistent *with a communication function γ if it satisfies, for all $\lambda_1, \lambda_2, \lambda_3 \in \mathcal{L}$ such that $\gamma(\lambda_1, \lambda_2) = \lambda_3$:*

1. *$npc_\ell(\lambda_1) \cup npc_\ell(\lambda_2) = npc_\ell(\lambda_3)$; and*
2. *$afc_\ell(\lambda_1) \cup afc_\ell(\lambda_2) = afc_\ell(\lambda_3)$.*

Example 6.30. *Consider the specification of Peterson's algorithm, it is straightforward to verify that the $\mathcal{C}(Pet)$-assignment (npc_ℓ, afc_ℓ) presented in Example 6.23 is consistent with the communication function γ. Consider, by way of example, the equation*

$$\gamma(r\_asgn^{\top}_{RA}, s\_asgn^{\top}_{RA}) = asgn^{\top}_{RA} \ ,$$

which is part of the definition of γ. We confirm as follows that indeed the conditions of Definition 6.29 are satisfied:

$$\begin{aligned}npc_\ell(r\_asgn^{\top}_{RA}) \cup npc_\ell(s\_asgn^{\top}_{RA}) = \{\text{L}, \text{RRL}\} = npc_\ell(asgn^{\top}_{RA}), \\ afc_\ell(r\_asgn^{\top}_{RA}) \cup afc_\ell(s\_asgn^{\top}_{RA}) = \{\text{L}, \text{RRL}\} = afc_\ell(asgn^{\top}_{RA}).\end{aligned}$$

If $npc_\ell : \mathcal{L} \to 2^C$ associates with every label a subset of components in C and $C' \subseteq C$, then we denote by $\mathcal{L}(C')$ the *alphabet* of C', i.e.,

$$\mathcal{L}(C') = npc_\ell{}^{-1}(C') = \{\lambda \in \mathcal{L} \mid npc_\ell(\lambda) = C'\} \ ,$$

and by $\mathcal{A}(C')$ the *action alphabet* of C', i.e.,

$$\mathcal{A}(C') = \mathit{afc}_\ell{}^{-1}(C') = \{\lambda \in \mathcal{L} \mid \mathit{afc}_\ell(\lambda) = C'\} \ .$$

Note that by condition 2 of Definition 6.22 we have $\mathcal{A}(C') \subseteq \mathcal{A}$.

Theorem 6.31. *Let E be a signal-respecting sequential recursive specification, let P be a parallel-sequential process expression over E, and let $(\mathit{npc}_\ell, \mathit{afc}_\ell)$ be a $\mathcal{C}(P)$-assignment. If $\mathcal{L}(P|_\sigma) \subseteq \mathcal{L}(\{\sigma\})$ and $\mathcal{A}(P|_\sigma) \subseteq \mathcal{A}(\{\sigma\})$ for all $\sigma \in \mathcal{C}(P)$ and γ is signal-respecting and consistent with $(\mathit{npc}_\ell, \mathit{afc}_\ell)$, then $(\mathit{npc}_\ell, \mathit{afc}_\ell)$ satisfies the Equations* (6.4) *and* (6.5) *for every transition t reachable from P.*

Proof. Let t be a transition reachable from P. Then $\mathit{src}(t) = P'$ for some parallel-sequential process expression P' reachable from P. By Lemma 6.27 we have $\mathcal{C}(P') = \mathcal{C}(P)$ and we have $P'|_c$ is reachable from $P|_c$ for all $c \in \mathcal{C}(P)$. So, without loss of generality, we may assume that $\mathit{src}(t) = P$.

Let $\mathit{npc}(t) = \{\sigma_0, \dots, \sigma_n\}$. By Lemma 6.28, t has subderivations $t_0, \dots, t_n$ such that $\mathit{src}(t_i)$ is a sequential process expression and $\mathit{src}(t_i) = \mathit{src}(t)|_{\sigma_i}$ for $0 \leq i \leq n$, and $\ell(t) = \overline{\gamma}([\ell(t_0), \dots, \ell(t_n)])$. Since, for all $0 \leq i \leq n$, $\ell(t_i) \in \mathcal{L}(P|_{\sigma_i}) \subseteq \mathcal{L}(\{\sigma_i\})$, we have $\mathit{npc}_\ell(\ell(t_i)) = \{\sigma_i\}$, and hence, by condition 1 of Definition 6.29,

$$\mathit{npc}(t) = \{\sigma_i \mid 0 \leq i \leq n\} = \bigcup_{0 \leq i \leq n} \mathit{npc}_\ell(\ell(t_i)) = \mathit{npc}_\ell(\ell(t)) \ .$$

Since E is a signal-respecting recursive specification and $\mathit{src}(t_i)$ is a signal-respecting sequential process expression, by Lemma 6.25 t_i is a signal transition if, and only if, $\ell(t_i) \in \mathcal{S}$. Since, on the one hand, $\mathit{afc}_\ell(\ell(t_i)) = \emptyset$ for all $\ell(t_i) \in \mathcal{S}$, and, on the other hand, $\mathcal{L}(P|_\sigma) \cap \mathcal{A} \subseteq \mathcal{A}(\{\sigma\})$ we have

$$\begin{aligned} \mathit{afc}(t) &= \{\sigma_i \mid 0 \leq i \leq n \text{ and } \lambda_i \in \mathcal{A}\} \\ &= \bigcup \{\mathit{afc}_\ell(\ell(t_i)) \mid 0 \leq i \leq n \text{ and } \ell(t_i) \in \mathcal{A}\} = \mathit{afc}_\ell(\ell(t)) \ . \end{aligned}$$

This completes the proof of the theorem. □

If E, P, γ and $(\mathit{npc}_\ell, \mathit{afc}_\ell)$ satisfy the requirements of the preceding theorem, then the relation $\smile\!\!\bullet$ on labels given by $\lambda_1 \smile\!\!\bullet\, \lambda_2$ if, and only if, $\mathit{npc}_\ell(\lambda_1) \cap \mathit{afc}_\ell(\lambda_2) = \emptyset$ satisfies the requirements of Definition 6.18. So we get the following corollary.

Corollary 6.32. *Let E be a signal-respecting sequential recursive specification, let P be a parallel-sequential process expression over E, and let $(\mathit{npc}_\ell, \mathit{afc}_\ell)$ be a $\mathcal{C}(P)$-assignment such that $\mathcal{L}(P|_\sigma) \subseteq \mathcal{L}(\{\sigma\})$ and $\mathcal{A}(P|_\sigma) \subseteq \mathcal{A}(\{\sigma\})$ for all $\sigma \in \mathcal{C}(P)$ and γ is signal-respecting and consistent with $(\mathit{npc}_\ell, \mathit{afc}_\ell)$. Then the LTSC associated with P has a concurrency-consistent labelling.*

Example 6.33. *In Example 6.30 we have established that all the conditions of Corollary 6.32 are satisfied for E_{Pet}, γ, Pet and the $\mathcal{C}(\mathit{Pet})$-assignment $(\mathit{npc}_\ell, \mathit{afc}_\ell)$ defined in Example 6.23, so the LTSC associated with Pet has a concurrency-consistent labelling.*

6.7 Expressing Liveness

A mathematically rigorous method for establishing the correctness of a (finite model of a) system is by means of *model checking.* Given a process expression specifying a system, the behaviour of that system can be scrutinised by verifying which requirements, expressed in a modal logic, hold true and which ones fail to hold. Among the modal logics that can be used to express such requirements is the modal μ-calculus. This is one of the most expressive logics available, subsuming logics such as HML, LTL, CTL and CTL*, and it is typically used in tool suites for analysing labelled transition systems, such as the mCRL2 toolset [24] and CADP [42]. We recall this logic in Section 6.7.1.

Liveness requirements typically assert that (conditionally or unconditionally) something good must inevitably happen. Phrasing such properties in the modal μ-calculus is rather standard, but it is less clear whether the logic permits expressing liveness properties restricted to just paths only. This is partly due to the fact that justness is a predicate on paths, whereas the modal μ-calculus is a state-based formalism, and partly due to the 'dynamic' nature of justness, which checks along a path for enabledness of actions and their future elimination. In particular this dynamic nature rules out a 'static' encoding such as the one presented in [35] for dealing with fairness, as it assumes an *a priori* fixed—i.e., static—collection of constraints that need to hold infinitely often for a path to be fair.

We show that liveness requirements of the form 'along every just path, every a action is inevitably followed by a b action' can indeed be expressed in the modal μ-calculus. Other path-based properties can be defined along the same lines. We discuss the liveness property in Section 6.7.2.

6.7.1 The Modal μ-Calculus

The modal μ-calculus (L_μ) was already introduced in Section 2.3. Since we use an LTSC here we need to make slight adjustments. Let $(St, Tr, src, target, \ell, \smile\!\!\bullet)$ be a finite LTSC over $\mathcal{L}$ with a concurrency-consistent labelling. We proceed to give a denotational semantics for our logic by associating every formula φ with the subset $[\![\varphi]\!]_\vartheta \subseteq St$ of states in which it holds. We reuse Definition 2.12 but alter the definition of $[\![\langle\lambda\rangle\varphi]\!]_\vartheta$ and $[\![[\lambda]\varphi]\!]_\vartheta$ as follows:

$$\begin{aligned}
[\![\langle\lambda\rangle\varphi]\!]_\vartheta &= \{s \in St \mid \exists t \in Tr.\ s = src(t) \text{ and } \ell(t) = \lambda \text{ and} \\
&\qquad target(t) \in [\![\varphi]\!]_\vartheta\} \\
[\![[\lambda]\varphi]\!]_\vartheta &= \{s \in St \mid \forall t \in Tr.\ \text{if } s = src(t) \text{ and } \ell(t) = \lambda \text{ then} \\
&\qquad target(t) \in [\![\varphi]\!]_\vartheta\}
\end{aligned}$$

6.7.2 Expressing Liveness along Just Paths

There are different liveness properties that one might be interested in. A basic liveness property is 'whenever some non-blocking action **a** happens, then inevitably also **b** happens'; this property will be referred to as **a**-**b**-*liveness.* We generalise this notion to $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness: 'whenever ϕ holds and some action $\mathbf{a} \in \mathbf{A}$ happens, then inevitably also some action $\mathbf{b} \in \mathbf{B}$ happens or ψ becomes true'. A state is said to satisfy $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness exactly when all paths emanating from that state satisfy $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness. Note that we can express **a**-**b**-liveness with $(\{\mathbf{a}\},\top)$-$(\{\mathbf{b}\},\bot)$-liveness. The generalisation from **a**-**b**-liveness to $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness is first described in this work; it is a necessary extension to accommodate liveness properties encountered in the verification of EULYNX interfaces. We require that ϕ and ψ are closed L_μ formulas. An L_μ formula that asserts that this property holds along all paths in a given deadlock-free LTS is the following

$$\nu X.\,(\bigwedge_{\lambda\in\mathcal{A}}[\lambda]X \wedge (\phi \implies (\bigwedge_{\mathbf{a}\in\mathbf{A}}([\mathbf{a}]\mu Y.\,(\psi \vee \bigwedge_{\lambda\in\mathcal{A}\setminus\mathbf{B}}[\lambda]Y))))).$$

Restricting $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness to *just* paths requires that somehow the concept of justness is woven into this formula. We explain in several steps how this can be achieved.

In order to facilitate our reasoning, we consider the dual problem of characterising an $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness violation along *some* just path. While this problem is technically equally difficult, it is conceptually simpler since we are now only concerned with constructing a formula that describes the *existence* of a just path. Notice that a (just) path constitutes a violation to $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness precisely when (1) this path has a suffix starting at a state s', reached by an $\mathbf{A}$-labelled transition from a state in which ϕ holds, along which no action $\mathbf{b} \in \mathbf{B}$ ever takes place and ψ never holds and (2) the path is just.

Our approach to characterising states that admit a violating path (should one exist) is based on the following observations. We can characterise states that admit a just, $(\mathbf{B},\psi)$-free path. Given any such state, we can characterise the states reaching it via a path ending with an $\mathbf{A}$-labelled transition, i.e. any **a**-labelled transition with $\mathbf{a} \in \mathbf{A}$. Any just path can be prefixed by an arbitrary finite path, resulting in a new just path (see Proposition 6.35 below).

For the remainder of this section we fix a finite LTSC $(\mathit{St}, \mathit{Tr}, \mathit{src}, \mathit{target}, \ell, \smile\!\!\bullet)$ with a concurrency-consistent labelling.

Definition 6.34. *Let $\pi = s_0t_1s_1t_2\ldots$ be a finite or infinite path. The path is $(\mathbf{B},\psi)$-free when $s \notin [\![\psi]\!]$ for every state s on π and $\ell(t) \notin \mathbf{B}$ for every transition t on π.*

The justness rephrasing of Proposition 6.19 requires one to reason about the enabled actions of a state. Let $\mathsf{En}(s)$ be the set of enabled non-blocking actions: $\mathsf{En}(s) = \{\lambda \in \overline{\mathcal{B}} \mid \exists t \in \mathit{Tr} : \mathit{src}(t) = s \text{ and } \ell(t) = \lambda\}$.

Proposition 6.35. *Let π be a $\mathcal{B}$-just path. Then the path $s_0t_1s_1\ldots t_n\pi$ is $\mathcal{B}$-just.*

Proof. Let $\pi' = s_0t_1s_1t_2\ldots t_n\pi$ be a path such that π is $\mathcal{B}$-just, and let s_π be the starting state of π. To prove that π' is $\mathcal{B}$-just, by Prop. 6.19 it suffices to prove that for all states on π' any enabled non-blocking action is eliminated in the suffix starting in that state. Suppose s is a state on π' and $\lambda \in \mathsf{En}(s)$. We distinguish two cases.

- Case s does not occur in the prefix $s_0 t_1 s_1 t_2 \dots t_n$. Then s occurs in π and since π is $\mathcal{B}$-just, λ is eliminated in the suffix of π (and therefore also in the suffix of π'), starting in s.

- Case s occurs in the prefix $s_0 t_1 s_1 t_2 \dots t_n$. Towards a contradiction, assume that λ is not eliminated in the suffix of π' starting in s. Let t be the transition such that $\ell(t) = \lambda$ and $src(t) = s$. Since λ is not eliminated in the suffix of π' starting in s and s_π is reachable from s, by condition 2 of Def. 6.2, there must be an action transition u such that $src(u) = s_\pi$ and $\ell(t) = \lambda = \ell(u)$. But then $\lambda \in \mathsf{En}(s_\pi)$ and, since π is $\mathcal{B}$-just, λ is eliminated in π. Contradiction. Consequently, λ is eliminated in the suffix of π' starting in s. □

The suffixes of a just path are again just. This is formalised by the following proposition.

Proposition 6.36. *Let $\pi = s_0 t_1 s_1 t_2 \dots$ be a finite or infinite path. If π is $\mathcal{B}$-just then also any suffix of π is $\mathcal{B}$-just.*

Proof. Let π be a $\mathcal{B}$-just path and let π' be a suffix of π. Pick some state s in π' and an action $\lambda \in \mathsf{En}(s)$. Since s is in π', s is also in π. Consequently, λ must be eliminated by some action in the suffix of π starting at s. Since s is in π', the suffix of π starting at s also is a suffix of π'. □

We next lift the notion of just path to the level of states: a *state* is just whenever it is the start of a just path. Note that we are interested in characterising states that admit a just path constituting an $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness violation; such paths must have suffixes that are void of $\mathbf{B}$-actions and for which in every state ψ does not hold. For this reason, we characterise the set of states admitting a liveness violating $\mathcal{B}$-just path in the following definition.

Definition 6.37. *We define $\mathcal{J}(\mathcal{A} \setminus \mathbf{B}, \psi)$ as follows:*

$$\mathcal{J}(\mathcal{A} \setminus \mathbf{B}, \psi) = \{s \in St \mid \textit{there is a } (\mathbf{B}, \psi)\textit{-free } \mathcal{B}\textit{-just path } \pi \textit{ starting in } s\}$$

As we explained at the beginning of this section, we tackle our problem in two steps. First we show that formula $\mathsf{invariant}$, see Table 6.3, characterises the states that admit a just path along which ψ never holds and no $\mathbf{B}$-action ever happens; i.e., those are essentially the states in the set $\mathcal{J}(\mathcal{A} \setminus \mathbf{B}, \psi)$. Then we continue by characterising states that have a path ending in a state where ϕ holds and an $\mathbf{A}$-labelled transition to a state in $\mathcal{J}(\mathcal{A} \setminus \mathbf{B}, \psi)$ is enabled. That is, we show that the formula that characterises the set of states that admit an $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness violation are exactly those states satisfying formula $\mathsf{violate}$ of Table 6.3.

Before we prove our claim that $\mathsf{invariant}$ exactly characterises states admitting just, $(\mathbf{B}, \psi)$-free paths we first prove an auxiliary lemma. This auxiliary lemma claims that $\mathsf{elim}(\lambda)$ captures exactly those states that have a $(\mathbf{B}, \psi)$-free path that eliminates action λ, and leads to a set of states represented by Y.

$$
\begin{array}{ll}
\mathsf{violate} = & \mu W.\ (\,(\phi \land \bigvee_{\lambda \in \mathbf{A}} \langle\lambda\rangle \mathsf{invariant}) \lor \bigvee_{\lambda \in \mathcal{A}} \langle\lambda\rangle W\,) \\[2ex]
\mathsf{invariant} = & \nu Y.\ \neg\psi \land \bigwedge_{\lambda \in \overline{\mathcal{B}}} ((\langle\lambda\rangle\top \Rightarrow \mathsf{elim}(\lambda)) \\[2ex]
\mathsf{elim}(\lambda) = & \mu Q.\ \neg\psi \land (\,\bigvee_{\lambda' \in \#\lambda \setminus \mathbf{B}} \langle\lambda'\rangle Y \lor \bigvee_{\lambda' \in \mathcal{A} \setminus (\#\lambda \cup \mathbf{B})} \langle\lambda'\rangle Q\,) \\[2ex]
\end{array}
$$

where $\#\lambda = \{\lambda' \mid \lambda \not\smile^{\bullet} \lambda'\}$

Table 6.3: Template formula that characterises the set of states that admit a just path violating $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness. Subformula $\mathsf{invariant}$ characterises the set of states that admit a just, $(\mathbf{B}, \psi)$-free path. The user provides the sets $\mathcal{A}$ and $\overline{\mathcal{B}}$, the relation $\smile^{\bullet}$, the formulas ϕ and ψ and the sets of actions $\mathbf{A}$ and $\mathbf{B}$ to instantiate/generate the formula for checking a concrete LTSC.

Lemma 6.38. *For all environments ϑ, states $s \in St$, actions $\lambda \in \mathcal{A}$ and sets $\mathcal{F} \subseteq St$ such that $\mathcal{F} \subseteq [\![\neg\psi]\!]$, we have $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{F}]}$ if, and only if, a state in $\mathcal{F}$ can be reached from s via a finite $(\mathbf{B}, \psi)$-free path ending with an action that eliminates λ and all preceding actions do not eliminate λ.*

Proof. Let ϑ be an arbitrary environment. Let $\mathcal{R}$ denote the set of states that emit a finite $(\mathbf{B}, \psi)$-free path ending in a state of $\mathcal{F}$, ending with an action that eliminates λ and on which all preceding actions do not eliminate λ. We first show, by mutual set inclusion, that $\mathcal{R}$ is a fixed point of the transformer T_{elim} defined below:

$$
\begin{aligned}
&T_{\mathsf{elim}}(\mathcal{F}') = [\![\neg\psi]\!] \cap (\\
&\{s \in St \mid \exists t \in Tr : src(t) = s \land \ell(t) \in \#\lambda \setminus \mathbf{B} \land target(t) \in \mathcal{F}\} \ \cup \\
&\qquad \{s \in St \mid \exists t \in Tr : src(t) = s \land \ell(h) \in \mathcal{A} \setminus (\#\lambda \cup \mathbf{B}) \land target(t) \in \mathcal{F}'\})
\end{aligned}
$$

- Pick an arbitrary state $s \in \mathcal{R}$. We must show that $s \in T_{\mathsf{elim}}(\mathcal{R})$. Let π be a path that witnesses $s \in \mathcal{R}$. Note that this path has at least one transition since it ends with a transition eliminating λ. For every state s' in π it holds that $s' \in [\![\neg\psi]\!]$, so also $s \in [\![\neg\psi]\!]$. We proceed with a case distinction on the length of the path π. Suppose π has length 1: $\pi = s\,t_0\,s_1$. Then by the definition of $\mathcal{R}$ we have that $\ell(t_0) \in \#\lambda \setminus \mathbf{B}$ and $s_1 \in \mathcal{F}$. We conclude that $s \in T_{\mathsf{elim}}(\mathcal{R})$. Next, suppose that the path π has length greater than 1: $\pi = s\,t_0\,\pi'$. The path π' is then also a $(\mathbf{B}, \psi)$-free path ending with an action that eliminates λ and ending in a state of $\mathcal{F}$. Hence, the first state in π', say s_1, is also in $\mathcal{R}$. Moreover, by the definition of $\mathcal{R}$, t_0 does not eliminate λ: $\ell(t_0) \in \mathcal{A} \setminus (\#\lambda \cup \mathbf{B})$. We conclude that $s \in T_{\mathsf{elim}}(\mathcal{R})$.

- Pick an arbitrary state $s \in T_{\mathsf{elim}}(\mathcal{R})$. We must show that $s \in \mathcal{R}$. By the definition of T_{elim} we derive that $s \in [\![\neg\psi]\!]$. We proceed with a case distinction. Suppose there is some transition t_0 such that $src(t_0) = s$, $target(t_0) = s_1 \in \mathcal{F}$ and $\ell(t_0) \in \#\lambda \setminus \mathbf{B}$. By our assumption that $\mathcal{F} \subseteq [\![\neg\psi]\!]$ we have that $s_1 \in [\![\neg\psi]\!]$. The path $s\,t_0\,s_1$ is a $(\mathbf{B},\psi)$-free path ending with an action that eliminates λ; hence $s \in \mathcal{R}$. Next, suppose there is some transition t_0 such that $src(t_0) = s$, $target(t_0) \in \mathcal{R}$ and the label does not eliminate λ: $\ell(t_0) \in \mathcal{A} \setminus (\#\lambda \cup \mathbf{B})$. Then $target(t_0)$ has some path π witnessing that $target(t_0) \in \mathcal{R}$. The path $s\,t_0\,\pi$ then witnesses that $s \in \mathcal{R}$.

We conclude that $\mathcal{R}$ is indeed a fixed point of T_{elim}. We next show that $\mathcal{R}$ is the least fixed point of T_{elim}; that is, for any $\mathcal{F}'$ satisfying $T_{\mathsf{elim}}(\mathcal{F}') = \mathcal{F}'$, we have $\mathcal{R} \subseteq \mathcal{F}'$. Let $\mathcal{F}'$ be a fixed point of T_{elim}, and choose $s \in \mathcal{R}$. Our aim is to show that $s \in \mathcal{F}'$. There is some path $\pi = s\,t_0\,s_1 \ldots s_{n-1}\,t_{n-1}\,s_n$ witnessing that $s \in \mathcal{R}$. We proceed by induction on the length of the path π. The base case is the case where $n = 1$. The path π is then just $s\,t_0\,s_1$ with $\ell(t_0) \in \#\lambda \setminus \mathbf{B}$ and $\{s, s_1\} \subseteq [\![\neg\psi]\!]$. From this we can conclude that $s \in T_{\mathsf{elim}}(\mathcal{F}')$ and since $\mathcal{F}'$ is a fixed point of T_{elim} also $s \in \mathcal{F}'$. For the step case we assume that $s_1 \in \mathcal{R}$ implies that $s_1 \in \mathcal{F}'$ (induction hypothesis). It follows from the definition of $\mathcal{R}$ and the fact that $s \in \mathcal{R}$ that also $s_1 \in \mathcal{R}$. Hence, by the induction hypothesis we have that $s_1 \in \mathcal{F}'$. We will proceed to show that then also $s \in \mathcal{F}'$. Since $s \in \mathcal{R}$ we know that $s \in [\![\neg\psi]\!]$ and since t_0 is not the last transition (the path π has at least length 2) it follows that $\ell(t_0) \in \mathcal{A} \setminus (\#\lambda \cup \mathbf{B})$. Then $s \in T_{\mathsf{elim}}(\mathcal{F}')$ and since $\mathcal{F}'$ is a fixed point of T_{elim} also $s \in \mathcal{F}'$. □

We continue by substantiating the claim that invariant characterises the states admitting just, $(\mathbf{B},\psi)$-free paths. For the sake of conciseness, let $\mathcal{J}$ be a shorthand for $\mathcal{J}(\mathcal{A} \setminus \mathbf{B}, \psi)$. The following lemma states that invariant exactly characterises the set of states $\mathcal{J}$.

Lemma 6.39. *For all $s \in St$ we have $s \in \mathcal{J}$ if and only if $s \in [\![\mathsf{invariant}]\!]$.*

Proof. Let ϑ be an arbitrary environment. We first show, by showing mutual set inclusion, that $\mathcal{J}$ is a fixed point of the transformer $T_{\mathsf{invariant}}$ defined below:

$$T_{\mathsf{invariant}}(\mathcal{F}) = \bigcap_{\lambda \in \mathcal{A}} \{s \in St \mid s \in [\![\neg\psi]\!] \wedge (\lambda \in \mathsf{En}(s) \Rightarrow s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{F}]})\}$$

- Pick an arbitrary state $s \in \mathcal{J}$. Let π be a path that witnesses $s \in \mathcal{J}$. Then for every state s' in π it holds that $s' \in [\![\neg\psi]\!]$. Pick an arbitrary action $\lambda \in \mathcal{A}$ and assume that $\lambda \in \mathsf{En}(s)$. We must show that $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{J}]}$ holds. From the fact that π witnesses $s \in \mathcal{J}$, we obtain that there must be some first transition t on π such that $\lambda \not\smile^{\bullet} \ell(t)$, i.e., $\ell(t) \in \#\lambda$, and by Prop. 6.36, $target(t) \in \mathcal{J}$. With the help of Lemma 6.38 we can conclude the desired $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{J}]}$.
- Pick a state $s \in T_{\mathsf{invariant}}(\mathcal{J})$. Suppose $\mathsf{En}(s) = \emptyset$. Then state s itself is a just path with $s \in [\![\neg\psi]\!]$ and hence $s \in \mathcal{J}$. Next, suppose $\mathsf{En}(s) \neq \emptyset$ and let

$\lambda \in \mathsf{En}(s)$. Then also $s \in \llbracket \neg\psi \rrbracket$ and $s \in \llbracket \mathsf{elim}(\lambda) \rrbracket_{\vartheta[Y:=\mathcal{J}]}$. By Lemma 6.38, there must be some $(\mathbf{B}, \psi)$-free finite path $s = s_0\, t_0\, s_1\, t_1 \ldots t_j\, s_{j+1}$ such that transition t_j eliminates λ and $s_{j+1} \in \mathcal{J}$. By Prop. 6.35, then also the path witnessing $s_{j+1} \in \mathcal{J}$, prefixed with $s_0\, t_0\, s_1\, t_1 \ldots t_j$, is a just path witnessing $s \in \mathcal{J}$.

We conclude that, indeed, $\mathcal{J}$ is a fixed point of $T_{\mathsf{invariant}}$. We next show that $\mathcal{J}$ is the greatest fixed point of $T_{\mathsf{invariant}}$; that is, for any $\mathcal{F}$ satisfying $T_{\mathsf{invariant}}(\mathcal{F}) = \mathcal{F}$, we have $\mathcal{F} \subseteq \mathcal{J}$. Let $\mathcal{F}$ be a fixed point of $T_{\mathsf{invariant}}$, and choose $s \in \mathcal{F}$. Our aim is to show that $s \in \mathcal{J}$. First, observe that since $\mathcal{F}$ is a fixed point of $T_{\mathsf{invariant}}$ and $s \in \mathcal{F}$, we can conclude $s \in T_{\mathsf{invariant}}(\mathcal{F})$.

We construct a just, $(\mathbf{B}, \psi)$-free path starting in state s by eliminating all actions enabled in s in an arbitrary but fixed order as follows. Let L denote the set of enabled actions in s. In case $L = \emptyset$, the state s itself witnesses $s \in \mathcal{J}$ and we are done. Otherwise, fix a total ordering $<$ on L. Pick the least action $\lambda \in L$. Since $s \in T_{\mathsf{invariant}}(\mathcal{F})$, also $s \in \llbracket \mathsf{elim}(\lambda) \rrbracket_{\vartheta[Y:=\mathcal{F}]}$ holds. Consequently, by Lemma 6.38, there is a finite $(\mathbf{B}, \psi)$-free path $s_0\, t_0\, s_1\, t_1 \ldots t_j\, s_\lambda$ such that transition t_j eliminates λ and $s_\lambda \in \mathcal{F}$. Denote the set of enabled actions in s_λ by L_λ. Note that L_λ contains at least those actions of L that are not eliminated on any path from s to s_λ (it may, however, contain actions that are eliminated on *some* path from s to s_λ, but these actions were then not eliminated on *all* paths from s to s_λ witnessing $s \in \llbracket \mathsf{elim}(\lambda) \rrbracket_{\vartheta[Y:=\mathcal{F}]}$). We now repeat this construction by choosing the least $\lambda' \in \{\lambda'' \in L_\lambda \cap L \mid \lambda < \lambda''\}$, leading to a state $s_{\lambda'}$, etcetera, until we have constructed a finite path that eliminates all obligations in L and ends in a state $s' \in \mathcal{F}$. Note that this construction terminates since $|L| \leq |\mathcal{L}| < \infty$.

This means that for any state $s \in \mathcal{F}$, we can construct a finite $(\mathbf{B}, \psi)$-free path to another state in $\mathcal{F}$, say s', such that all actions from $\mathsf{En}(s)$ are eliminated on that path; and for all states along this path all non blocking actions that are not yet eliminated are still enabled in s' due to Corollary 6.15. Since this holds invariantly for all states in $\mathcal{F}$, this construction can be repeated to yield a finite $(\mathbf{B}, \psi)$-free just path or (in case it can be continued indefinitely) an infinite $(\mathbf{B}, \psi)$-free just path starting in s. Hence, $s \in \mathcal{J}$ and therefore $\mathcal{F} \subseteq \mathcal{J}$. $\square$

We illustrate the correspondence between $\mathsf{invariant}$ and $\mathcal{J}$ on the example we provided earlier.

Example 6.40. *Reconsider Example 6.4, in which Alice drinks coffee and subsequently eats a croissant, Bob is engaged in a series of phone calls, and none of their activities interfere; see the following LTSC:*

$$\begin{tikzcd}
s_0 \arrow[loop above, "phone"] \arrow[r, "coffee"'] & s_1 \arrow[loop above, "phone"] \arrow[r, "croissant"'] & s_2 \arrow[loop above, "phone"]
\end{tikzcd}$$

Suppose we claim that whenever Alice orders coffee, she eventually also orders a croissant. A counterexample to such a claim consists of a just path that contains *a coffee event but is free of croissant actions following this coffee event. A state admits such a violating, croissant-less path if and only if it satisfies formula* $\mathsf{invariant}$.

We argue that in this case, s_1 does not satisfy formula $\mathsf{invariant}$. *To this end, we first show that s_1 does not satisfy* $\mathsf{elim}(\mathit{croissant})$. *Notice that the set $\#\mathit{croissant} \setminus \{\mathit{croissant}\} = \{\mathit{coffee}\}$, while the set $\mathcal{A} \setminus (\#\mathit{croissant} \cup \{\mathit{croissant}\})$ is the set $\{\mathit{phone}\}$. Formula* $\mathsf{elim}(\mathit{croissant})$ *therefore effectively holds in s_1 if and only if formula $\langle \mathit{phone} \rangle Q$ holds in state s_1. Due to the self-loop, this is the case exactly when state s_1 satisfies* $\mathsf{elim}(\mathit{croissant})$. *Since this chain of reasoning must be continued indefinitely and we are looking for the least solution to Q, we must conclude that s_1 does not satisfy* $\mathsf{elim}(\mathit{croissant})$. *As an immediate consequence we find that s_1 also does not satisfy* $\mathsf{invariant}$ *since croissant is one of the enabled actions in that state. Observe that this is in line with the fact that s_1 does not admit a croissant-free just path.*

We now return to the original problem of characterising those states that have a just path that violates $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness. So far, we have established that formula $\mathsf{invariant}$ characterises those states that admit a $(\mathbf{B}, \psi)$-free, just path. A state that admits a path violating $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness is therefore one that admits a finite path that, via an $\mathbf{A}$-labelled transition from a state in which ϕ holds, leads to a state satisfying $\mathsf{invariant}$. Given the similarities with the formula for elim, we claim, without further proof, that formula $\mathsf{violate}$ indeed describes the set of states that admit an $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness violating just path.

Theorem 6.41. *Let $(\mathit{St}, \mathit{Tr}, \mathit{src}, \mathit{target}, \ell, \smile\!\!\bullet)$ be a finite LTSC with a concurrency-consistent labelling. Then all just paths starting in state $s \in \mathit{St}$ satisfy $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness if, and only if, $s \notin [\![\mathsf{violate}]\!]$.*

Example 6.42. *We continue our previous example, showing that, indeed, the claim that whenever Alice orders coffee, she eventually also orders a croissant, holds true in state s_0.*

We find that s_0 satisfies $\mathsf{violate}$ *if, and only if, it satisfies* $\langle \mathit{coffee} \rangle \mathsf{invariant}$, $\langle \mathit{coffee} \rangle \mathsf{violate}$, $\langle \mathit{phone} \rangle \mathsf{violate}$, *or* $\langle \mathit{croissant} \rangle \mathsf{violate}$. *Notice that there is no croissant action enabled in s_0, so s_0 cannot satisfy* $\langle \mathit{croissant} \rangle \mathsf{violate}$. *In order for s_0 to satisfy* $\langle \mathit{phone} \rangle \mathsf{violate}$, *we require s_0 to again satisfy* $\mathsf{violate}$. *Like before, such a cyclic chain of reasoning does not permit us to conclude that s_0 satisfies* $\mathsf{violate}$. *Therefore, the only way to show that s_0 satisfies* $\mathsf{violate}$ *is to show that s_0 satisfies* $\langle \mathit{coffee} \rangle \mathsf{invariant}$. *But as we may conclude from our previous example, also this will fail since s_1 does not satisfy* $\mathsf{invariant}$, *which is required when we are to conclude that s_1 satisfies* $\mathsf{invariant}$. *We can therefore conclude that state s_0 does not satisfy* $\mathsf{violate}$. *Since the LTSC has a concurrency-consistent labelling, we may conclude by Theorem 6.41 that our liveness claim holds and Alice enjoys a croissant after drinking coffee.*

Example 6.43. *In Example 6.30, we concluded that all the conditions of Corollary 6.32 are satisfied for E_{Pet}, γ, Pet and the $\mathcal{C}(\mathit{Pet})$-assignment $(\mathit{npc}_\ell, \mathit{afc}_\ell)$, so* *the LTSC associated with Pet has a concurrency-consistent labelling. As a consequence, by Theorem 6.41 we can therefore conclude that the formula in Table 6.3, with $\mathcal{B} = \{$***noncritA, noncritB***$\}$, **A** $= \{$***noncritA***$\}$, $\phi = \top$, **B** $= \{$***critA***$\}$ and $\psi = \bot$ expresses* **noncritA-critA**-*liveness.*

6.8 Automated Liveness Analysis in mCRL2

A complete mCRL2 specification of Peterson's algorithm can be found in the Zenodo repository. The recursive specification E_{Pet} presented in Section 6.3 served as the starting point and the reader will easily recognise it under the mCRL2 keyword `proc`. That the mCRL2 specification looks somewhat more involved than the specification presented in Section 6.3 is because we have used some convenient extra features of mCRL2. Before we comment on these extra features, we emphasise that the use of these features is by no means essential. We could have also verified liveness for all just paths with the mCRL2 toolset with a specification that almost literally corresponds to the one presented in Section 6.3.

In an mCRL2 specification, labels can be parametrised with data, defined by means of an algebraic specification. In our specification we have included an enumerated type of which the elements correspond to the labels of Peterson's specification. This allows us to define, in a natural way, the functions npc_ℓ and afc_ℓ as mappings `npc` and `afc`, respectively, on the `Label` datatype. We then define a predicate `interfere(a,a')` that evaluates to true if, and only if, $npc_\ell($`a`$) \cap afc_\ell($`a'`$) \neq \emptyset$ using the mappings `npc` and `afc`. In a similar vein, a predicate `blocking(a)` defines whether `a` is blocking or not.

The correspondence between labels and the data values representing them is achieved by turning the labels of *Pet* into *multi-actions*, 'labelling' the original actions with a parametrised action `label(<action>)`, where `<action>` identifies the original action. For instance, we represent the label **critA** using data value `a_critA`. In the equation defining `procA`, we have, instead of the occurrence of **critA** appearing in *procA* in Section 6.3, a multi-action `critA|label(a_critA)`. We can then choose to either hide the labels of the form `label(<label>)`, or hide the labels representing those in the specification of Peterson's algorithm in Section 6.3. The former allows us to generate a labelled transition system that is identical to that associated with *Pet*; the latter yields a labelled transition system in which transitions are labelled with actions of the form `label(<label>)`.

The toolset accepts the first-order modal μ-calculus of [59], which generalises the logic L_μ. With the labels available as a datatype and using the predicates `interfere` and `blocking`, we can express the formula expressing liveness for all just paths as an almost direct instantiation (with $\{$**noncritA**$\}$ for **A**, $\top$ for ϕ, $\{$**critA**$\}$ for **B**, and $\bot$ for ψ) of the formula in Table 6.3. The formula we have used to verify that the mCRL2 specification of Peterson's algorithm satisfies the required liveness property is listed below. The extra features of mCRL2 described above facilitate writing the generalised disjunctions and conjunctions as existential and universal quantifications. Note, however, that, since the quantifications are over finite sets, they can be replaced by finite disjunctions and conjunctions.

```
¬ μ W. (
    <label(a_noncritA)>(
      ν Y. ∀ a:Label.(val(¬blocking(a)) && <label(a)>⊤) =>
        μ Q. (
          (∃ a':Label.val(interfere(a,a')
            ∧ (a' ≠ a_critA)) ∧ <label(a')>Y)
        ∨
          (∃ a':Label.val(¬interfere(a,a')
            && (a' ≠ a_critA)) && <label(a')>Q )
        )
    )
  ∨ ∃ a:Label.<label(a)>W)
```

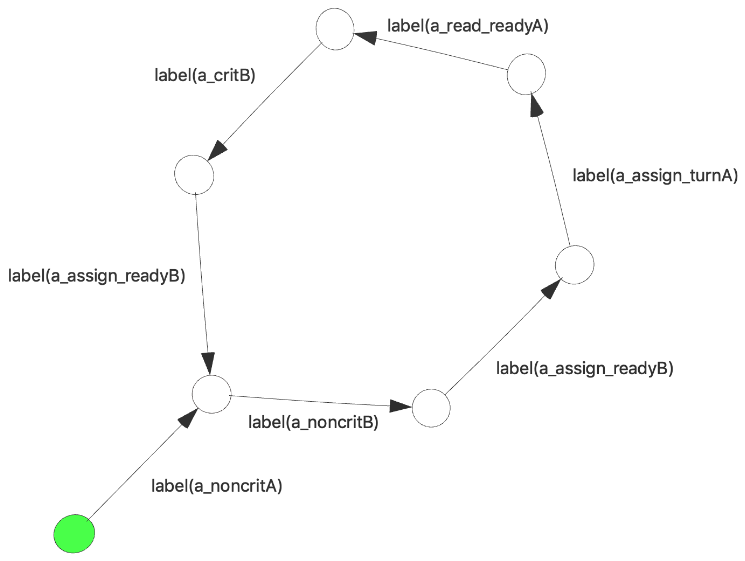


Figure 6.2: Non-just liveness counterexample generated by the mCRL2 toolset.

Verifying whether the mCRL2 specification of Peterson's algorithm satisfies **noncritA-critA**-liveness requires under half a second using the toolset and results in an affirmative verdict.[2] This once more confirms the manual correctness proof of [33]. If we modify the specification of the mapping `afc` by including `c_ReadyA` in `afc(a_read_readyA)`, `c_ReadyB` in `afc(a_read_readyB)`, and `c_Turn` in both `afc(a_read_turnA)` and `afc(a_read_turnB)`, then the toolset produces the counterexample shown in Figure 6.2. Note that the modification corresponds to not treating these actions as signals and that the counterexample represents the non-just path discussed in Section 6.3.

[2]The mCRL2 sources can be found in the *academic* example directory of the mCRL2 repository, which can be obtained from `https://github.com/mCRL2org/mCRL2`, revision `b45856d9a`. Additionally they can be found in the Zenodo repository [12].

6.9 Application for mCRL2 Models Derived from SysML Models

In the context of EULYNX interfaces we are also interested in verifying liveness requirements; in Chapter 5 discussing case studies, we have presented several such requirements. In particular we conjectured that requirement REQ_PDI_6 and REQ_P_1, which did not hold, might hold under a justness assumption. In Section 6.9.1 we will present how we can verify liveness properties with a justness assumption for mCRL2 models which are instantiations of the generic SysML semantics model from Chapter 3. Additionally, we will revisit the aforementioned requirements in Section 6.9.2. The models and requirements, along with instructions on how to replicate our results, can be found in the Zenodo repository [12].

6.9.1 Formulating $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness

To instantiate the $(\mathbf{A},\phi)$-$(\mathbf{B},\psi)$-liveness formula (Table 6.3) we need to supply the set of action labels $\mathcal{A}$, the set of signal labels $\mathcal{S}$, the set of blocking actions $\mathcal{B}$ and the relation $\smile\!\!\bullet$. The pair of action sets $\mathbf{A}$ and $\mathbf{B}$, and the formulas ϕ and ψ are specific to a requirement and will be presented in the next section.

Defining the Labels

The notation and terminology around actions and labels has in this chapter been slightly different than in Chapter 2, mostly because of the multi-action semantics of mCRL2. We will work towards expressing which multi-actions are in $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{B}$. The set of action labels, using the terminology and notation of Chapter 2, is as follows:

$$\begin{aligned}\Lambda = \{&\texttt{discardEvent}, \texttt{selectMultiStep}, \texttt{executeBehaviour},\\ &\texttt{executeStep}, \texttt{inState}, \texttt{inEventPool}, \texttt{resetVariables},\\ &\qquad\qquad\qquad\qquad\qquad \texttt{varVal}, \texttt{send}, \texttt{receive}\}\end{aligned}$$

Restating some definitions from Chapter 2: the set of actions $\{a(\mathbf{e}) \mid a \in \Lambda, \mathbf{e} \in \mathbb{D}_a\}$ is denoted by Ψ, where $\mathbb{D}_a$ denotes the semantic domain of the sort of action label a. The set of all multi-sets over Ψ is denoted by Ω.

Note that for τ labelled transitions we cannot determine which component(s) are affected. We can therefore not hide the internal actions related to transition selection.

The set $\mathcal{A}$ and $\mathcal{S}$ will be defined as subsets of Ω. Not every multi-action in Ω should be included in $\mathcal{A}$ or $\mathcal{S}$, some multi-actions are not possible due to the allow operator. In particular, except for the `send` and `receive` actions, every allowed multi-action consists of only one action. The following action labels are used in signals: `inEventPool`, `resetVariables`, and `varVal`. The rest are used in actions.

$$\mathcal{S} = \{\wr a(\mathbf{e}) \int \in \Omega \mid a \in \{\texttt{inState}, \texttt{inEventPool}, \texttt{varVal}\} \text{ and } \mathbf{e} \in \mathbb{D}_a\}$$

$$\begin{aligned}\mathcal{A} = \{\omega \in \Omega \setminus \mathcal{S} \mid\ & (\omega = \wr a(\mathbf{e}) \int \text{ with } \mathbf{e} \in \mathbb{D}_a \text{ and } a \in \{\texttt{discardEvent},\\ & \texttt{selectMultiStep}, \texttt{executeBehaviour},\\ & \texttt{executeStep}, \texttt{resetVariables}\})\\ & \vee (\underline{\omega}(\texttt{send}) = 1 \text{ and } \#\omega = \underline{\omega}(\texttt{send}) + \underline{\omega}(\texttt{receive}) > 1)\}\end{aligned}$$

Defining the Concurrency Relation

The mCRL2 model contains a sort `CompName` listing the names of components. Some of the component names identify the environment. Whether there is one environment name or multiple depends on how the generic model is instantiated. Our automated translation tool introduces two `CompName`s for every component: one to be used in a `StateMachine` process and one to be used by the environment when communicating with that component. Let $\mathcal{C}$ denote the set of `CompName`s used in a `StateMachine` process. Every action in Ψ has the name of a component among its data parameters. This component name is either in $\mathcal{C}$ or a name used by the environment. We define the mapping $\mathit{Comp} : \Psi \to \mathcal{C} \cup \{\textsc{env}\}$ as $\mathit{Comp}(a(\mathbf{e})) = c$ if $c \in \mathcal{C}$ and it occurs in $\mathbf{e}$ and $\mathit{Comp}(a(\mathbf{e})) = \textsc{env}$ otherwise.

We will only consider inputs from the environment to be blocking. We could also consider certain `selectMultiStep` actions to be blocking when they are not dependent on a timeout or event. Since it is not clear when such transitions are blocking we will consider them non-blocking. Note that violate becomes weaker as we consider more actions to be blocking.

$$\mathcal{B} = \{\omega \in \mathcal{A} \mid \exists_{\texttt{send}(\vec{\boldsymbol{d}}) \in \omega} \mathit{Comp}(\texttt{send}(\vec{\boldsymbol{d}})) = \textsc{env}\}$$

As before, we specify the relation $\smile\!\!\bullet$ by specifying the sets of necessary participants npc_ℓ and affected components afc_ℓ for each label in $\mathcal{A}$. For two labels λ_1 and λ_2 we have that $\lambda_1 \smile\!\!\bullet \lambda_2$ if, and only if, $\mathit{npc}_\ell(\lambda_1) \cap \mathit{afc}_\ell(\lambda_2) = \emptyset$.

We distinguish two cases which cover all the labels in $\mathcal{A}$:

1. For the transition labels related to SysML transition selection and execution a single component is affected. For every $\omega \in \mathcal{A}$, if $\omega = \wr a(\vec{\boldsymbol{d}}) \int$ with

   $$\begin{aligned}a \in \{&\texttt{discardEvent}, \texttt{selectMultiStep}, \texttt{executeBehaviour},\\ &\texttt{executeStep}, \texttt{resetVariables}\}\end{aligned}$$

   then we define $\mathit{npc}_\ell(\omega) = \mathit{afc}_\ell(\omega) = \{\mathit{Comp}(a(\vec{\boldsymbol{d}}))\}$.

2. For communications between state machines both the sender and receivers are affected. For every $\omega \in \mathcal{A}$, if for every $a \in \underline{\omega}$ we have that $a = \texttt{send}$ or $a = \texttt{receive}$ then we define

   $$\mathit{npc}_\ell(\omega) = \mathit{afc}_\ell(\omega) = (\bigcup_{a(\vec{\boldsymbol{d}}) \in \omega} \{\mathit{Comp}(a(\vec{\boldsymbol{d}}))\}) \setminus \{\textsc{env}\}.$$

With these definitions the LTSC is concurrency-consistent. Every signal is a selfloop and the condition that for all transitions $t, u \in \mathit{Tr}$ we have that $t \mathrel{\smile\!\!\bullet} u$ if, and only if, $\ell(t) \mathrel{\smile\!\!\bullet} \ell(u)$ is trivially satisfied since we defined $\mathrel{\smile\!\!\bullet}$ in terms of the labels.

Replacing all actions in the model with `label` actions as described in Section 6.3 is difficult for our models. Especially because of our use of `send|receive` multi-actions. The occurrence of such multi-actions should be replaced with a single `label` action containing the `CompNames` of the sender and all receivers. Instead we provide an alternative template formula for liveness requirements tailored to FormaSig models, which can be found below. The template formula is kept (somewhat) concise by lifting the elimination of actions to the elimination of components: if a non-blocking action is enabled for some component then that component will eventually perform an action. This transformation is sound because an action λ is eliminated exactly when one of the participating components performs an action. In the formula we use some overloading: **A** and **B** are action formulas[3] capturing the sets **A** and **B**. In the formula, **ENV** should be replaced with a list of `CompNames` that are used to denote the environment.

```
¬(μW. (%violate
    <⊤>W ∨ (φ ∧ <A>(
      %invariant
      νY. ¬ψ ∧ (∀ c:CompName.(
          val(¬(c ∈ [ENV_sp,ENV_fp,ENV_pe51])) => ((<(
          (∃ e:Event. discardEvent(c, e))
          ∨ (∃ e:Event, sc:StateConfig. selectMultiStep(c,e,sc))
          ∨ (∃ sc:StateConfig. executeStep(c,sc))
          ∨ (executeBehaviour(c))
          ∨ (resetVariables(c))
          ∨ (∃ v:Value. ∃ p,p2:VarName. ∃ c2:CompName.
            val(¬(c2 ∈ ENV))
            ∧ send(CompPortPair(c2,p2),v)|receive(CompPortPair(c,p),v))
          ∨ (∃ v:Value,p,p2:VarName,c2:CompName.
            receive(CompPortPair(c2,p2),v)|send(CompPortPair(c,p),v))
        )>⊤) =>
        %elim
        μ Q. ¬ψ ∧ (
          (<¬B ∧ (
            (∃ e:Event. discardEvent(c, e))
              ∨ (∃ e:Event, sc:StateConfig. selectMultiStep(c,e,sc))
              ∨ (∃ sc:StateConfig. executeStep(c,sc))
              ∨ (executeBehaviour(c))
              ∨ (resetVariables(c))
              ∨ (∃ v:Value. ∃ p,p2:VarName. ∃ c2:CompName.
               send(CompPortPair(c2,p2),v)|receive(CompPortPair(c,p),v))
              ∨ (∃ v:Value. ∃ p,p2:VarName. ∃ c2:CompName.
               receive(CompPortPair(c2,p2),v)|send(CompPortPair(c,p),v))
            )>Y)
          ∨ ( ∃ c2:CompName. val(c2 ≠ c) ∧ <¬B ∧ (
              (∃ e:Event. discardEvent(c2, e))
              ∨ (∃ e:Event, sc:StateConfig. selectMultiStep(c2,e,sc))
              ∨ (∃ sc:StateConfig. executeStep(c2,sc))
              ∨ (executeBehaviour(c2))
              ∨ (resetVariables(c2))
              ∨ (∃ v:Value. ∃ p,p2:VarName. ∃ c3:CompName.
                val(¬(c3 == c))
                ∧ send(CompPortPair(c3,p2),v)|receive(CompPortPair(c2,p),v))
              ∨ (∃ v:Value. ∃ p,p2:VarName. ∃ c3:CompName.
```

[3] https://mcrl2.org/web/user_manual/language_reference/mucalc.html

```
               val(¬(c3 == c))
               ∧ receive(CompPortPair(c3,p2),v)|send(CompPortPair(c2,p),v))
           )>Q )
      ))))))))
```

We have now defined everything needed to instantiate an $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness formula for FormaSig models, except of course for $\mathbf{A}$, $\mathbf{B}$, ϕ and ψ.

6.9.2 Revisiting Requirements

Let us consider requirement REQ_PDI_6 for the generic EULYNX interface (see Section 5.4): 'When F_EST_EfeS_SR signals it is not ready for a connection, the object controller will move to not ready for connection and only reattempts a connection after receiving a message ready for connection'. This requirement consists of two parts, the latter of which is cover by REQ_PDI_6_1. We will focus on the requirement that the object controller needs to move to a state where it is not ready for a connection. We can capture it in a $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness formula with the following definitions. The values for $\mathbf{A}$ and ϕ express that the component BEQ_eest signals that the object controller is not ready for a connection whilst the PDI connection is established.

```
A = {send(CompPortPair(BEQ_eest,T18_Not_Ready_For_PDI_Connection),
                      Value_Pulse_Pack([]))
           |receive(CompPortPair(BEQ_seec,T18_Not_Ready_For_PDI_Connection),
                      Value_Pulse_Pack([]))}
φ = <inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>⊤
```

By picking $\mathbf{B}$ to be empty we express that always eventually ψ must hold. The formula ψ expresses that the system is in a deadlock due to full event pools *or* the component BEQ_seec is in the state NOT_READY_FOR_CONNECTION *or* BEQ_seec is in the state PDI_CONNECTION_IMPERMISSIBLE.

```
B = {}
ψ = (νX. <eventPoolFull>⊤ ∧  ⋀   [λ]X)
                            λ∈A
      ∨ <inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>⊤
      ∨ <inState(BEQ_seec,PDI_CONNECTION_IMPERMISSIBLE)>⊤
```

The formula holds for the EULYNX generic interface mCRL2 model. The original formula without the justness assumption did not.

Similarly, consider requirement REQ_P_1 for the EULYNX point interface (see Section 5.5): 'The object controller must report changes in position, in particular when it was in an end position and it no longer is'. We capture this requirement as an $(\mathbf{A}, \phi)$-$(\mathbf{B}, \psi)$-liveness formula with the following definitions. The values for $\mathbf{A}$ and ϕ express that the object controller perceives that the point is no longer in an end position whilst it was previously in the left or right end position.

```
A = {receive(CompPortPair(BEQ_pe51, D21_PM1_Position),
                 Value_Custom(STR_NO_END_POSITION))
         |send(CompPortPair(ENV_pe51, D21_PM1_Position),
                 Value_Custom(STR_NO_END_POSITION))}
φ = <inState(BEQ_pe51, ALL_LEFT)>⊤ ∨ <inState(BEQ_pe51, ALL_RIGHT)>⊤
```

The values for $\mathbf{B}$ and ψ express that always eventually it is reported to the interlocking that the point is no longer in an end position *or* a deadlock due to full event pools is reached *or* the connection is lost.

```
B = {receive(CompPortPair(ENV_sp, T20_Point_Position),
                Value_Pulse_Pack([VarValuePair(DT20_Point_Position,
                  Value_Custom(STR_NO_END_POSITION))]))
        |send(CompPortPair(BEQ_sp, T20_Point_Position),
                Value_Pulse_Pack([VarValuePair(DT20_Point_Position,
                  Value_Custom(STR_NO_END_POSITION))]))}
ψ = (νX. <eventPoolFull>⊤ ∧  ⋀   [λ]X)
                            λ∈A
        ∨ <inState(BEQ_pe51,WAITING_FOR_INITIALISING)>⊤
        ∨ [inState(BEQ_fp,PDI_CONNECTION_ESTABLISHED)]⊥
        ∨ [inState(BEQ_sp,PDI_CONNECTION_ESTABLISHED)]⊥
```

This formula holds for the point mCRL2 model, whereas the original without the justness assumption did not.

6.10 Conclusions

To facilitate the automated verification of liveness properties, we have proposed a notion of concurrency-consistent labelling for labelled transition system with concurrency together with a formulation of justness in terms of states and actions. We have presented sufficient conditions on a process specification in a calculus with ACP-style communication that guarantee that the associated labelled transition system with concurrency has a concurrency-consistent labelling. Moreover, for LTSCs with a concurrency-consistent labelling we have shown how to formalise a liveness property under justness assumptions in the modal μ-calculus.

We have built on the firm foundation laid by Van Glabbeek in [47], but had to slightly deviate from it to enable a special treatment of signal transitions in a regular process calculus. Furthermore, we essentially relied on the ACP-style communication mechanism in our calculus.

We have achieved our original goal of verifying two liveness requirements for EULYNX interfaces with a justness assumption. In both cases, justness proved to be strong enough to rule out unrealistic liveness violating computations.

We have also applied the theory to Peterson's mutual exclusion algorithm and have shown that it can be specified in such a way that the associated LTSC has a concurrency-consistent labelling. Using the mCRL2 toolset we were able to verify that the specification satisfies the required liveness property for all just paths. We conjecture that similar specifications can be realised for the generalisation of Peterson's algorithm to N processes [104], and for Lamport's bakery algorithm [77]; it remains to confirm liveness properties for all just paths for these specifications with the mCRL2 toolset.

We see several directions in which our current work can be extended. For example, it would be useful to automate the verification of the syntactic conditions that guarantee that a specification induces an LTSC that has a concurrency-consistent labelling. A more challenging task is to identify to which extent the fragment of the process calculus can be extended without losing the guarantee that the LTSCs associated with expressions in that fragment have a concurrency-consistent labelling. We believe it may even be possible to check sufficient conditions for the LTSC to have a concurrency-consistent labelling by phrasing appropriate modal μ-calculus formulas. Finally, open issues in the context of justness are the definitions of component-preserving variants of behavioural equivalences. A variant of strong bisimilarity [100], called enabling preserving bisimilarity [44] has already been presented by Van Glabbeek et al., but to our knowledge there are no variants of other behavioural equivalences in the linear time – branching time spectrum [46]. A particularly interesting equivalence would be divergence-preserving branching bisimilarity [50], because it deals with abstraction and is the coarsest congruence included in branching bisimilarity that distinguishes livelock from deadlock and is compatible with parallel composition [51].

Chapter 7

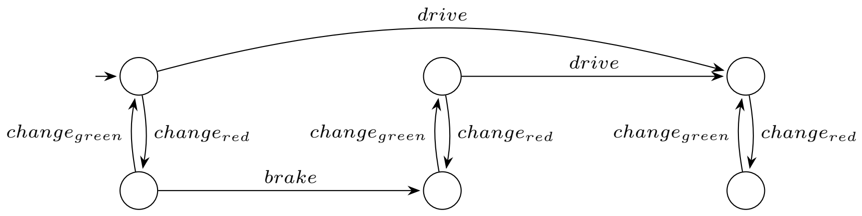
Global Variables in Process Algebras

You can see God from anywhere if your mind is set to love and obey Him.

A. W. Tozer

Communication between parallel components in real world systems takes many forms: packets over a network, inter-process communication, communication via shared memory, communication over a bus, etcetera. Process algebras usually offer an abstract message passing feature. Not all forms of communication fit well in a message passing paradigm, in particular, global variables and other forms of shared memory do not fit in well. In some cases it would be desirable to have global variables as first class citizens. To illustrate this we introduce a small example.

Example 7.1. *Consider a traffic light and a car approaching a junction. If the light is green the car performs an action* drive *and moves past the junction. If the light is red the car performs an action* brake *and stops. Once the traffic light is green again the car performs the action* drive*. The specification should result in the following LTS.*



It would be natural to model the car and the traffic light as two parallel com-

ponents. The car and the traffic light need to communicate so that the car can make a decision to drive or brake depending on the current state of the traffic light. Typically, such global information is modelled by introducing an extra parallel component that maintains the global information, in this case the colour of the traffic light. However, taking that approach we obtain a different LTS, which has an extra transition modelling the communication of car and traffic light with the extra component. Moreover, one must take care that decisions to drive or brake are made on the basis of up-to-date information, e.g., by implementing a protocol that locks the additional component. In many cases it is realistic that the observed value is no longer up to date and in some cases we are also interested in analysing the consequences of this. In other cases however, we might want to abstract from such complications. When the information is constantly available to the observers, as is the case with a traffic light, we have even more reason to not introduce separate transitions communicating the global information.

In some process algebras it is possible to define a communication function that specifies that a drive action is the result of a communication between two actions of parallel components, e.g. a drive_if_green action from the car and a signal_green action from the traffic light. This is somewhat unnatural as the traffic light does not really actively participate in the driving of the car. Moreover, if we introduce a second car that only wants to drive when the traffic light is red we would need to change the communication function, even though the communication of information does not essentially change. It would be better to let the colour of the traffic light be in a global variable. In that way the behaviour of the traffic light and cars acting upon information from the traffic light are independent, we obtain a better separation of concerns.

We are also interested in global variables from the viewpoint of verification. The semantics of mCRL2 (and almost every other process algebra) is action-based and thus abstracts from the contents of a state; states are only distinguished by their transitions. The logic (the modal μ-calculus in the case of the mCRL2 toolset) also only refers to transitions. For models that are derived from state based formalisms (such as SysML) it can be desirable to also consider the contents of a state. As we saw in the case studies chapter, we added selfloops 'annotating' states with data (e.g. `inState` selfloops showing the current SysML state configuration), which we used to formulate certain requirements. Adding such selfloops is somewhat strange as these transitions do not actually represent a step the system can take. It would be better if the states could directly be annotated with data and if we could refer to that data in the logic.

Example 7.2. *Reconsider the example of the traffic light and suppose that we want to check the property "The action 'drive' only takes place when the traffic light is green". Formulating this property in a logic such as the Hennessy-Milner Logic (HML) or the modal μ-calculus is a bit awkward. You need to express that between any drive action and the last preceding $change_{green}$ action there has not been a $change_{red}$ action, but there could be any number of other actions. In the modal μ-calculus extended with regular formulas (as used by the mCRL2 toolset)* *this can be expressed in the following way, where we assume the signal is green in the initial state.*

$$[\top * \ . \ change_{red} \ . \ (\neg change_{green}) * \ . \ drive]\bot$$

It would be more convenient if states were annotated with the value of the traffic light which could be referenced in the logic. Suppose that we have some operator $(v = d)$ *which holds for states where variable* v *has value* d*. Next suppose that there is some global variable light which can take the values green and red. We could then formulate the property concisely in the following way.*

$$\langle drive\rangle\top \implies (light = green)$$

The goal in this chapter is to set a first step towards the integration of global variables in mCRL2. With this direction in mind we propose and study (i) a simple process calculus with global variables (ii) a modal logic that can refer to the values of global variables and (iii) an encoding allowing us to use existing tools. We propose a simple process calculus where every component of the system can access the current values of global variables directly. We define appropriate notions of equivalence for our process calculus. Our first contribution is an extension of HML with two new operators that is strong enough to differentiate non-equivalent process expressions. Our second contribution is a transformation of LTSs with global data embedded in the state to LTSs with 'plain' states and an encoding of our extended logic in standard HML. This encoding is such that the translated formula holds for a state in the transformed LTS if, and only if, the original formula holds for the corresponding state in the original LTS. For our final contribution we study shared (non-global) data and how this can be tracked using a dedicated operator in the process algebra.

This chapter is organised as follows. In Section 7.1 we define a simple process algebra with global variables. In Section 7.2 we give appropriate notions of equivalence for our process algebra. We continue by defining an extension of HML in Section 7.3 and relating it to our equivalence notions. In Sections 7.4 and 7.5 we show how to check extended HML formulas by transforming both the LTS and the formula, and how this approach relates to our use of selfloops for the verification of EULYNX interfaces. Our final contribution, on the topic of scoped variables, is presented in Section 7.6. In Section 7.7 we relate our work to earlier publications on global data in process algebras. Finally, in Section 7.8 we conclude the chapter.

7.1 A Simple Process Algebra with Global Variables

In this section we introduce a process algebra with global variables and its semantics. In this chapter we assume, for convenience, a single data domain D. We use $\mathbb{V}$ to denote a finite set of global variable names.

We presuppose a set of actions $\mathcal{A}$ and define a set of transition labels $\mathcal{L} \overset{\text{def}}{=} \mathcal{A} \cup \{\alpha(v,d) \mid v \in \mathbb{V} \wedge d \in D\}$. The special action α denotes the assignment of a variable. In this chapter we consistently use Latin letters ($a, b, \ldots \in \mathcal{A}$) to denote action labels and Greek letters ($\lambda \in \mathcal{L}$) to denote transition labels. We also presuppose a set of process names $\mathcal{N}$. The set of process expressions $\mathcal{P}$ is generated by the following grammar containing *action prefix, inaction, choice, parallelism, encapsulation, recursion and conditionals*:

$$P := \lambda.P \mid \delta \mid P + P \mid P \parallel P \mid \partial_B(P) \mid X \mid (v = d) \rightarrow P,$$

where $\lambda \in \mathcal{L}$, $B \subseteq \mathcal{A}$, $X \in \mathcal{N}$, $v \in \mathbb{V}$ and $d \in D$. Inaction, δ, is similar to the process constant 0 in, for example, CCS [89] and TCP [1]. Our process algebra supports recursion; we define a recursive specification E defining the process names. Let a recursive equation be an equation of the form $X \overset{\text{def}}{=} t$ with $X \in \mathcal{N}$ and t a process expression in $\mathcal{P}$. A recursive specification contains one recursive equation $X \overset{\text{def}}{=} t$ for every $X \in \mathcal{N}$. Every recursive specification should be guarded. This means that every occurrence of a process name in t is in the scope of an action prefix. For communication between parallel processes we use an ACP style communication function. We presuppose a binary communication function on the set of actions, i.e., a partial function $\gamma : \mathcal{A} \times \mathcal{A} \rightharpoonup \mathcal{A}$ that is commutative and associative. We only allow handshakes (communication between two parties): if for some action labels a and b it is the case that $c = \gamma(a,b)$ then $\gamma(c,d)$ is undefined for every d.

Let $\mathcal{V}$ be the set of all functions $\mathbb{V} \rightarrow D$, i.e. the set of all valuations. Let $V \in \mathcal{V}$; we denote by $V[v \mapsto d]$ the assignment defined, for all $v' \in \mathbb{V}$ by:

$$V[v \mapsto d](v') = \begin{cases} d & \text{if } v' = v \\ V(v') & \text{if } v' \neq v \end{cases}$$

We now want to associate an LTS with the process algebra. As the behaviour of a process expression depends on the valuation of global variables, a state is a pair $\langle P, V \rangle$ of a process expression P and a valuation function V.

Definition 7.3. *We associate the LTS ($\mathcal{P} \times \mathcal{V}, \mathcal{L}, \rightarrow$) to our process algebra. The transition relation $\rightarrow$ is the least relation on states satisfying the rules below.*

$$\frac{}{\langle a.P, V\rangle \xrightarrow{a} \langle P, V\rangle} \qquad \frac{}{\langle \alpha(v,d).P, V\rangle \xrightarrow{\alpha(v,d)} \langle P, V[v \mapsto d]\rangle}$$

$$\frac{\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle}{\langle (v = d) \to P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle} V(v) = d \qquad \frac{\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle}{\langle X, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle} X \stackrel{def}{=} P$$

$$\frac{\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle}{\langle P + Q, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle} \qquad \frac{\langle Q, V\rangle \xrightarrow{\lambda} \langle Q', V'\rangle}{\langle P + Q, V\rangle \xrightarrow{\lambda} \langle Q', V'\rangle}$$

$$\frac{\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle}{\langle P \parallel Q, V\rangle \xrightarrow{\lambda} \langle P' \parallel Q, V'\rangle} \qquad \frac{\langle Q, V\rangle \xrightarrow{\lambda} \langle Q', V'\rangle}{\langle P \parallel Q, V\rangle \xrightarrow{\lambda} \langle P \parallel Q', V'\rangle}$$

$$\frac{\langle P, V\rangle \xrightarrow{a} \langle P', V\rangle \qquad \langle Q, V\rangle \xrightarrow{b} \langle Q', V\rangle}{\langle P \parallel Q, V\rangle \xrightarrow{c} \langle P' \parallel Q', V\rangle} \gamma(a, b) = c$$

$$\frac{\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle}{\langle \partial_B(P), V\rangle \xrightarrow{\lambda} \langle \partial_B(P'), V'\rangle} \lambda \notin B$$

Note that we only allow processes to synchronise on actions and not on assignments. This design decision was made since assignments change the valuation function, whereas actions cannot change the valuation. When two processes synchronise on assignments of the same variable then it is not clear what the resulting effect on the value of the variable should be.

The LTS contains all states associated to the process algebra, which is infinitely large. In this chapter we regularly draw the reachable states in the LTS from some specific state, which we call an *LTS fragment.*

Example 7.4. *Consider the interaction between a car and a traffic light controller (TLC). The TLC sets the colour of a traffic light which the driver of the car acts upon. There is one global variable t and the data domain consists of two elements $D = \{green, red\}$. The recursive specification consists of two process equations, given below.*

$$Car \stackrel{def}{=} ((t = green) \to drive.\delta) \ + \ ((t = red) \to brake.((t = green) \to drive.\delta))$$

$$TLC \stackrel{def}{=} ((t = green) \to \alpha(t, red).TLC) \ + \ ((t = red) \to \alpha(t, green).TLC)$$

Using the derivation rules we can derive the LTS fragment reachable from $\langle CAR \parallel TLC, V\rangle$, where $V(t) = green$. Note that this LTS fragment is isomorphic to the one presented in Example 7.1, modulo the different names for the transitions changing the global variable.

$$\begin{tikzcd}
\langle Car \,\|\, TLC, V\rangle \arrow[rr, bend left, "drive"] \arrow[d, shift left, "\alpha(t, red)"] & \langle ((t = green) \rightarrow drive.\delta \;\|\, TLC, V\rangle \arrow[r, "drive"] \arrow[d, shift left, "\alpha(t, red)"] & \langle \delta \,\|\, TLC, V\rangle \arrow[d, shift left, "\alpha(t, red)"] \\
\langle Car \,\|\, TLC, V[t \mapsto red]\rangle \arrow[u, shift left, "\alpha(t, green)"] \arrow[r, "brake"] & \langle ((t = green) \rightarrow drive.\delta \;\|\, TLC, V[t \mapsto red]\rangle \arrow[u, shift left, "\alpha(t, green)"] & \langle \delta \,\|\, TLC, V[t \mapsto red]\rangle \arrow[u, shift left, "\alpha(t, green)"]
\end{tikzcd}$$

7.2 Behavioural Equivalences

Behavioural equivalences are an essential concept in the field of process algebra. They enable us to reason about the equivalence of states and process expressions. This is critical to prove the soundness of axioms and correctness of state space reduction techniques. Note that strong bisimilarity (Definition 2.9) and branching bisimilarity (Definition 2.10) can still be applied in the context of global variables. They may however, not be the best fit. We will examine behavioural equivalence relations described in the literature that explicitly take the data in states in consideration.

The values of global variables may be essential to the modelled system and we may want to refer to them during verification. Strong bisimulation does not distinguish states based on the valuation of global variables; strongly bisimilar states may not have the same valuation. State-based bisimilarity [90], defined below, also distinguishes states based on their valuation of global variables. States that are state-based bisimilar are also strongly bisimilar.

Definition 7.5. *State-based bisimilarity: A relation $\mathcal{R}_{sb} \subseteq (\mathcal{P} \times \mathcal{V}) \times (\mathcal{P} \times \mathcal{V})$ is a state-based bisimulation relation if and only if for all states $\langle P, V_1\rangle$ and $\langle Q, V_2\rangle$ and labels λ we have $(\langle P, V_1\rangle, \langle Q, V_2\rangle) \in \mathcal{R}_{sb}$ implies that $V_1 = V_2$ and*

- *for all process expressions P' and valuation functions V': $\langle P, V_1\rangle \xrightarrow{\lambda} \langle P', V'\rangle$ implies there exists a process expression Q' such that $\langle Q, V_2\rangle \xrightarrow{\lambda} \langle Q', V'\rangle$ and $(\langle P', V'\rangle, \langle Q', V'\rangle) \in \mathcal{R}_{sb}$,*
- *for all process expressions Q' and valuation functions V': $\langle Q, V_2\rangle \xrightarrow{\lambda} \langle Q', V'\rangle$ implies there exists a process expression P' such that $\langle P, V_1\rangle \xrightarrow{\lambda} \langle P', V'\rangle$ and $(\langle P', V'\rangle, \langle Q', V'\rangle) \in \mathcal{R}_{sb}$.*

Two states $\langle P, V_1\rangle$ and $\langle Q, V_2\rangle$ are state-based bisimilar, denoted by $\langle P, V_1\rangle \,\underline{\leftrightarrow}_{sb}\, \langle Q, V_2\rangle$, if, and only if, there exists a state-based bisimulation relation $\mathcal{R}_{sb}$ such that $(\langle P, V_1\rangle, \langle Q, V_2\rangle) \in \mathcal{R}_{sb}$.

It is desirable that a behavioural equivalence is a congruence: if process expressions P and Q are behaviourally equivalent it should imply that they are interchangeable in any larger process expression and paired with any valuation. State-based bisimilarity is not defined at the level of process expressions though, but on states. Below we provide the definition of *process-congruence* [90], which takes into account that states contain data.

Definition 7.6. *Process-congruence: A relation $\sim \subseteq (\mathcal{P} \times \mathcal{V}) \times (\mathcal{P} \times \mathcal{V})$ is a* process-congruence *with respect to an n-ary process function f if and only if for all process expressions $\{(p_i, q_i) \in \mathcal{P} \times \mathcal{P} \mid 0 \leq i < n\}$ and for all $V \in \mathcal{V}$ such that $\langle p_i, V\rangle \sim \langle q_i, V\rangle$ (for all $0 \leq i < n$), we then also have that $\langle f(p_0, \ldots p_{n-1}), V\rangle \sim \langle f(q_0, \ldots q_{n-1}), V\rangle$. The relation $\sim$ is a process-congruence for a process algebra if, and only if, it is a process-congruence with respect to all the operators of the signature of the process algebra.*

State-based bisimilarity and strong bisimilarity are not a process-congruence for our process algebra, which we demonstrate with the following example.

Example 7.7. *Consider process expressions $P = (v = 0) \to a.\delta$ and $Q = a.\delta$. Note that P and Q are simply abbreviations of process expressions, not process names. Let V map a global variable v to 0 and $D = \{0, 1\}$. The fragments of the LTS reachable from $\langle P, V\rangle$ and $\langle Q, V\rangle$ are shown in Figure 7.1.*

$$\begin{tikzcd}
\langle P, V\rangle \arrow[rrr, "a"] & & & \langle \delta, V\rangle \\
\langle Q, V\rangle \arrow[rrr, "a"] & & & \langle \delta, V\rangle
\end{tikzcd}$$

Figure 7.1: Two LTS fragments depicting the states reachable from $\langle P, V\rangle$ and $\langle Q, V\rangle$

The states $\langle P, V\rangle$ and $\langle Q, V\rangle$ are clearly state-based bisimilar. We might be tempted to think that P and Q are interchangeable but that is not the case. The problem arises when we add a parallel component that can assign a different value to the global variable. Let us consider the process expression $R = \alpha(v, 1).\delta$. The fragments of the LTS reachable from $\langle P \parallel R, V\rangle$ and $\langle Q \parallel R, V\rangle$ are shown in Figure 7.2. The states $\langle P \parallel R, V\rangle$ and $\langle Q \parallel R, V\rangle$ are not state-based bisimilar. Hence, state-based bisimilarity is not a process-congruence for our process algebra.

$$\begin{tikzcd}
\langle P \parallel R, V\rangle \arrow[r, "a"] \arrow[dr, "\alpha(v,1)"] & \langle \delta \parallel R, V\rangle \arrow[dr, "\alpha(v,1)"] & \\
& \langle P \parallel \delta, V[v \mapsto 1]\rangle & \langle \delta \parallel \delta, V[v \mapsto 1]\rangle
\end{tikzcd}$$

$$\begin{tikzcd}
\langle Q \parallel R, V\rangle \arrow[r, "a"] \arrow[dr, "\alpha(v,1)"] & \langle \delta \parallel R, V\rangle \arrow[dr, "\alpha(v,1)"] & \\
& \langle Q \parallel \delta, V[v \mapsto 1]\rangle \arrow[r, "a"] & \langle \delta \parallel \delta, V[v \mapsto 1]\rangle
\end{tikzcd}$$

Figure 7.2: Two LTS fragments depicting the states reachable from $\langle P \parallel R, V\rangle$ and $\langle Q \parallel R, V\rangle$

To reason about the equivalence of process expressions we use the notion of stateless-bisimilarity, as defined in [90] and stated below. In essence, stateless-bisimilarity relates process expressions that behave the same under any valuation. The deduction system of our process algebra is in process-tyft format from which it follows that stateless-bisimilarity is a congruence [90].

Definition 7.8. *Stateless-bisimilarity: A relation $\mathcal{R}_{sl} \subseteq \mathcal{P} \times \mathcal{P}$ is a stateless bisimulation relation if, and only if, for all labels λ and process expressions P and Q such that $(P, Q) \in \mathcal{R}_{sl}$ the following two conditions hold:*

- *for all process expressions P' and valuation functions $V, V' \in \mathcal{V}$: $\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle$ implies there exists a process expression Q' such that $\langle Q, V\rangle \xrightarrow{\lambda} \langle Q', V'\rangle$ and $(P', Q') \in \mathcal{R}_{sl}$,*

- *for all process expressions Q' and valuation functions $V, V' \in \mathcal{V}$: $\langle Q, V\rangle \xrightarrow{\lambda} \langle Q', V'\rangle$ implies there exists a process expression P' such that $\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle$ and $(P', Q') \in \mathcal{R}_{sl}$.*

Two process expressions P and Q are stateless bisimilar, denoted by $P \underline{\leftrightarrow}_{sl}\ Q$, if, and only if, there exists a stateless bisimulation relation $\mathcal{R}_{sl}$ such that $(P, Q) \in \mathcal{R}_{sl}$.

We can now examine equivalence on both the level of process expressions with stateless bisimilarity and on the level of states with state-based bisimilarity. For any two process expressions P and Q we have that if $P \underline{\leftrightarrow}_{sl}\ Q$ then also $\langle P, V\rangle \underline{\leftrightarrow}_{sb} \langle Q, V\rangle$ for all valuations $V \in \mathcal{V}$ [90].

7.3 Extending Hennessy-Milner Logic

In order to reason about properties of a process expression or system specification we define a logic. Standard HML [66] is insufficient for our purpose, for two reasons. The first reason is that we would like to refer to global variables in the logic making the formulation of some properties much more convenient, as explained in the Introduction. The second reason for extending the logic is that we want a correspondence between the logic and the behavioural equivalences. There is a nice correspondence between strong bisimilarity and HML: two states in an LTS are strongly bisimilar if, and only if, they satisfy the same HML formulas [66]. This correspondence is often called the Hennessy-Milner theorem. This theorem proves that a state space can be minimised modulo strong bisimulation whilst preserving all properties. We would like a similar correspondence between the logic and state-based and stateless bisimilarity.

We extend HML with a check operator $(v = e)$. This operator evaluates to true if and only if global variable v has value e. This results in the following syntax for our logic HML^{check}:

$$\phi := \top \mid \neg\phi \mid \phi \wedge \phi \mid \langle T \rangle \phi \mid (v = e)$$

where T is a non-empty finite set of transition labels, $v \in \mathbb{V}$ and $e \in D$. The dual operators can be formulated using negation.

$$\begin{array}{lcl} \bot & = & \neg\top \\ \phi \vee \psi & = & \neg(\neg\phi \wedge \neg\psi) \\ [T]\phi & = & \neg(\langle T \rangle \neg\phi) \end{array}$$

The semantics of the standard HML operators is as usual. Let ϕ be a formula, let $(S, L, \rightarrow)$ be an LTS. We inductively define the interpretation of ϕ, notation $[\![\phi]\!]$, where $[\![\phi]\!]$ contains all states $u \in S$ where ϕ is true.

$$\begin{array}{lcl} [\![\top]\!] & = & S \\ [\![\neg\phi]\!] & = & S \setminus [\![\phi]\!] \\ [\![\phi \wedge \phi']\!] & = & [\![\phi]\!] \cap [\![\phi']\!] \\ [\![\langle T \rangle \phi]\!] & = & \{u \in S \mid \exists u' \in [\![\phi]\!], \lambda \in T : u \xrightarrow{\lambda} u'\} \end{array}$$

The semantics of $(v = e)$ is only defined for the LTS, $(\mathcal{P} \times \mathcal{V}, \mathcal{L}, \rightarrow)$, where the states contain global variables.

$$[\![v = e]\!] \quad = \quad \{\langle P, V \rangle \in \mathcal{P} \times \mathcal{V} \mid V(v) = e\}$$

We proceed by working towards a relation between the behavioural equivalences and the logic. First, we introduce the notion of *image-finiteness.*

Definition 7.9. *Image-finiteness: A state s is image finite if, and only if, the set $\{t \mid s \xrightarrow{\lambda} t\}$ is finite for every label λ.*

Theorem 7.10, below, expresses that states are state-based bisimilar if and only if they satisfy the same HML^{check} formulas. Recall that we defined the reachability of states in Section 2.2.

Theorem 7.10. *Let $\langle P, V\rangle$ and $\langle Q, V\rangle$ be states in LTS $(\mathcal{P} \times \mathcal{V}, \mathcal{L}, \rightarrow)$ and let all states reachable from $\langle P, V\rangle$ and $\langle Q, V\rangle$ be image-finite. Then $\langle P, V\rangle \underline{\leftrightarrow}_{sb} \langle Q, V\rangle$ if and only if for all HML^{check} formulas ϕ we have that $\langle P, V\rangle \in [\![\phi]\!] \Leftrightarrow \langle Q, V\rangle \in [\![\phi]\!]$.*

Proof. We prove the two implications separately. To prove the implication from left to right assume $P \underline{\leftrightarrow}_{sb} Q$. We prove that for all HML^{check} formulas ϕ we have that $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$ by induction on the structure of ϕ.
Base cases

- $\phi = \top$: $[\![\top]\!] = \mathcal{P} \times \mathcal{V}$ so $\langle P, V\rangle \in [\![\top]\!]$ and $\langle Q, V\rangle \in [\![\top]\!]$.
- $\phi = (v = e)$: $[\![v = e]\!] = \{\langle R, V'\rangle \in \mathcal{P} \times \mathcal{V} \mid V'(v) = e\}$ so both $\langle P, V\rangle$ and $\langle Q, V\rangle$ are in $[\![v = e]\!]$ if and only if $V(v) = e$.

Step

- $\phi = \neg\phi'$: $[\![\neg\phi']\!] = (\mathcal{P} \times \mathcal{V}) \setminus [\![\phi']\!]$. By the induction hypothesis $\langle P, V\rangle \in [\![\phi']\!]$ if and only if $\langle Q, V\rangle \in [\![\phi']\!]$. Then also $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$.
- $\phi = \phi' \wedge \phi''$: $[\![\phi' \wedge \phi'']\!] = [\![\phi']\!] \cap [\![\phi'']\!]$. By the induction hypothesis $\langle P, V\rangle \in [\![\phi']\!]$ if and only if $\langle Q, V\rangle \in [\![\phi']\!]$ and $\langle P, V\rangle \in [\![\phi'']\!]$ if and only if $\langle Q, V\rangle \in [\![\phi'']\!]$. Then also $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$.
- $\phi = \langle T\rangle\phi'$: $[\![\langle T\rangle\phi']\!] = \{u \in \mathcal{P} \times \mathcal{V} \mid \exists u' \in [\![\phi']\!], \lambda \in T : u \xrightarrow{\lambda} u'\}$. As $P \underline{\leftrightarrow}_{sb} Q$ there exists some state $\langle P', V'\rangle$ such that $\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle$ if and only if there exists a state $\langle Q', V'\rangle$ such that $\langle Q, V\rangle \xrightarrow{\lambda} \langle Q', V'\rangle$. If such states and transition do not exist there is nothing left to prove. If they do exist then $P' \underline{\leftrightarrow}_{sb} Q'$. By the induction hypothesis $\langle P', V'\rangle \in [\![\phi']\!]$ if and only if $\langle Q', V'\rangle \in [\![\phi']\!]$. Hence, $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$.

For the implication from right to left, assume that $\langle P, V\rangle$ and $\langle Q, V\rangle$ satisfy exactly the same formulas in HML^{check}. We shall prove that $\langle P, V\rangle \underline{\leftrightarrow}_{sb} \langle Q, V\rangle$. To this end, note that it is sufficient to show that the relation

$$\mathcal{R}_{sb} = \{(\langle X, Z\rangle, \langle Y, Z\rangle) \mid \langle X, Z\rangle, \langle Y, Z\rangle \in \mathcal{P} \times \mathcal{V} \text{ and } \langle X, Z\rangle \text{ and } \langle Y, Z\rangle \\ \text{satisfy the same HML}^{check} \text{ formulas}\}$$

is a state-based bisimulation relation. Assume that $\langle T, V'\rangle \mathcal{R}_{sb} \langle U, V'\rangle$ and there exists some state $\langle T', V''\rangle$ and label λ such that $\langle T, V'\rangle \xrightarrow{\lambda} \langle T', V''\rangle$. We shall now argue that there is a state $\langle U', V''\rangle$ such that $\langle U, V'\rangle \xrightarrow{\lambda} \langle U', V''\rangle$ and $\langle T', V''\rangle \mathcal{R}_{sb} \langle U', V''\rangle$. Since $\mathcal{R}_{sb}$ is symmetric, this suffices to establish that $\mathcal{R}_{sb}$ is a state-based bisimulation relation.

Now assume, towards a contradiction, that there is no $\langle U', V'' \rangle$ such that $\langle U, V' \rangle \xrightarrow{\lambda} \langle U', V'' \rangle$ and $\langle U', V'' \rangle$ satisfies the same HML^{check} formulas as $\langle T', V'' \rangle$. Since $\langle U, V' \rangle$ is image-finite, the set of processes that $\langle U, V' \rangle$ can reach by performing a λ-labelled transition is finite, say $\{\langle U_1, V_1 \rangle, \ldots, \langle U_n, V_n \rangle\}$ with $n \in \mathbb{N}$. For every $i \in \{1 \ldots n\}$, there exists a formula ϕ_i such that $\langle T', V'' \rangle \in [\![\phi_i]\!]$ and $\langle U_i, V_i \rangle \notin [\![\phi_i]\!]$ or there exists a variable v such that $V_i(v) \neq V''(v)$. Note that if there exists a v such that $V_i(v) \neq V''(v)$ then there is also a distinguishing formula ϕ_i: $(v = V''(v))$.

We are now in a position to construct a formula that is satisfied by $\langle T, V' \rangle$ but not by $\langle U, V' \rangle$, contradicting our assumption that $\langle T, V' \rangle$ and $\langle U, V' \rangle$ satisfy the same formulas. The formula

$$\langle \{\lambda\} \rangle (\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$$

is satisfied by $\langle T, V' \rangle$ but not by $\langle U, V' \rangle$. □

Process expressions $a.\delta$ and $(v = 0) \to a.\delta$ are not stateless bisimilar but in the case that we have a valuation function V that maps v to 0 then states $\langle a.\delta, V \rangle$ and $\langle (v = 0) \to a.\delta, V \rangle$ cannot be distinguished using HML^{check}. A challenge here is that the logic is defined on states whereas stateless bisimilarity is defined on process expressions. We would like to be able to distinguish process expressions using the logic though, especially for systems where the global variables are not under direct control of the system but form a context in which the modelled system (the process expression) operates.

We introduce a *set* operator $\downarrow (v := e)$ to the logic, which sets the value of a global variable v to e. Depending on whether we include the check operator, the set operator or both, we will refer to the logic with HML^{check}, HML^{set} or HML$^{check+set}$, respectively. Again, the semantics of this new operator is only defined for the LTS, $(\mathcal{P} \times \mathcal{V}, \mathcal{L}, \to)$, where the states contain global variables.

$$[\![\downarrow (v := e)\phi]\!] \quad = \quad \{\langle P, V \rangle \in \mathcal{P} \times \mathcal{V} \mid \langle P, V[v \mapsto e] \rangle \in [\![\phi]\!]\}$$

Note that the set operator allows us to distinguish $\langle a.\delta, V \rangle$ and $\langle (v = 0) \to a.\delta, V \rangle$ even if $V(v) = 0$. The formula $\downarrow (v := 1)\langle \{a\} \rangle \top$ distinguishes them. We will use the notation $\downarrow (V)$, $V \in \mathcal{V}$, to set the value of all global variables to the value specified by V. This is a shorthand for a sequence of regular set operations. Note that the number of global variables is finite and the order of set operations is irrelevant in the sequence of set operations as each sets a different variable.

Before we can evaluate the correspondence between HML$^{check+set}$ and stateless bisimilarity we must lift the notions of reachability and image-finiteness to process expressions.

Definition 7.11. *(Reachability process expressions) Process expression P' is reachable from a process expression P, denoted by $P \to^* P'$, if, and only if, there are processes $P = P_0, \ldots, P_n = P'$ and labels $\lambda_0, \ldots, \lambda_{n-1}$ such that for all $0 \leq i < n$ there exists $V_i, V_i' \in \mathcal{V}$ such that $\langle P_i, V_i \rangle \xrightarrow{\lambda_1} \langle P_{i+1}, V_i' \rangle$.*

Definition 7.12. *(Image-finiteness process expressions) A process expression P is image finite if and only if for every valuation V the state $\langle P, V\rangle$ is image-finite.*

Theorem 7.13, below, states that process expressions are stateless bisimilar if and only if all states containing that process expression satisfy the same HML$^{check+set}$ formulas.

Theorem 7.13. *Let P and Q be process expressions such that all process expressions reachable from P and Q are image-finite. Then $P \underline{\leftrightarrow}_{sl} Q$ if and only if for all valuations $V \in \mathcal{V}$ and all HML$^{check+set}$ formulas ϕ we have that $\langle P, V\rangle \in [\![\phi]\!] \Leftrightarrow \langle Q, V\rangle \in [\![\phi]\!]$.*

Proof. We prove the two implications separately. To prove the implication from left to right assume $P \underline{\leftrightarrow}_{sl} Q$. We prove that for all HML$^{check+set}$ formulas ϕ and valuations V we have that $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$ by induction on the structure of ϕ.
Base cases

- $\phi = \top$: $[\![\top]\!] = \mathcal{P} \times \mathcal{V}$ so $\langle P, V\rangle \in [\![\top]\!]$ and $\langle Q, V\rangle \in [\![\top]\!]$.
- $\phi = (v = e)$: $[\![v = e]\!] = \{\langle R, V'\rangle \ \in \mathcal{P} \times \mathcal{V} \mid V'(v) = e\}$ so both $\langle P, V\rangle$ and $\langle Q, V\rangle$ are in $[\![v = e]\!]$ if and only if $V(v) = e$.

Step

- $\phi = \neg\phi'$: $[\![\neg\phi']\!] = (\mathcal{P} \times \mathcal{V}) \setminus [\![\phi']\!]$. By the induction hypothesis $\langle P, V\rangle \in [\![\phi']\!]$ if and only if $\langle Q, V\rangle \in [\![\phi']\!]$. Then also $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$.
- $\phi = \phi' \wedge \phi''$: $[\![\phi' \wedge \phi'']\!] = [\![\phi']\!] \cap [\![\phi'']\!]$. By the induction hypothesis $\langle P, V\rangle \in [\![\phi']\!]$ if and only if $\langle Q, V\rangle \in [\![\phi']\!]$ and $\langle P, V\rangle \in [\![\phi'']\!]$ if and only if $\langle Q, V\rangle \in [\![\phi'']\!]$. Then also $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$.
- $\phi = \langle T\rangle\phi'$: $[\![\langle T\rangle\phi']\!] = \{u \in \mathcal{P} \times \mathcal{V} \mid \exists u' \in [\![\phi']\!], \lambda \in T : u \xrightarrow{\lambda} u'\}$. As $P \underline{\leftrightarrow}_{sl} Q$ there exists some state $\langle P', V'\rangle$ such that $\langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle$ if and only if there exists a state $\langle Q', V'\rangle$ such that $\langle Q, V\rangle \xrightarrow{\lambda} \langle Q', V'\rangle$. If such states and transition do not exist there is nothing left to prove. If they do exist $P' \underline{\leftrightarrow}_{sl} Q'$. By the induction hypothesis $\langle P', V'\rangle \in [\![\phi']\!]$ if and only if $\langle Q', V'\rangle \in [\![\phi']\!]$. Hence, $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$.
- $\phi = \downarrow (v := e)\phi'$: $[\![\{\langle R, V'\rangle \in \mathcal{P} \times \mathcal{V} \mid \langle R, V'[v \mapsto e]\rangle \in [\![\phi]\!]\}]\!]$. Then $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle P, V[v \mapsto e]\rangle \in [\![\phi']\!]$ and $\langle Q, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V[v \mapsto e]\rangle \in [\![\phi']\!]$. By the induction hypothesis $\langle P, V[v \mapsto e]\rangle \in [\![\phi']\!]$ if and only if $\langle Q, V[v \mapsto e]\rangle \in [\![\phi']\!]$. Hence, $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$.

For the implication from right to left, assume that for all $V \in \mathcal{V}$ it holds that $\langle P, V\rangle$ and $\langle Q, V\rangle$ satisfy exactly the same formulas in HML$^{check+set}$. We shall prove that $P \underline{\leftrightarrow}_{sl} Q$. To this end, note that it is sufficient to show that the relation

$$\mathcal{R}_{sl} = \{(T,U) \mid T,U \in \mathcal{P} \text{ and for all } V \in \mathcal{V}, \langle T,V\rangle \text{ and } \langle U,V\rangle \text{ satisfy the same HML}^{check+set} \text{ formulas}\}$$

is a stateless bisimulation relation. Assume that $T\mathcal{R}_{sl}U$ and $\langle T,V'\rangle \xrightarrow{\lambda} \langle T',V''\rangle$ for some valuation V'. We shall now argue that there is a process U' such that $\langle U,V'\rangle \xrightarrow{\lambda} \langle U',V''\rangle$ and $T'\mathcal{R}_{sl}U'$. Since $\mathcal{R}_{sl}$ is symmetric, this suffices to establish that $\mathcal{R}_{sl}$ is a stateless bisimulation relation.

Now assume, towards a contradiction, that there is no $\langle U',V''\rangle$ such that $\langle U,V'\rangle \xrightarrow{\lambda} \langle U',V''\rangle$ and for all valuations $V \in \mathcal{V}$, $\langle U',V\rangle$ satisfies the same HML$^{check+set}$ formulas as $\langle T',V\rangle$. Since $\langle U,V'\rangle$ is image-finite, the set of processes that $\langle U,V'\rangle$ can reach by performing a λ-labelled transition is finite, say $\{\langle U_1,V_1\rangle,\ldots,\langle U_n,V_n\rangle\}$ with $n \in \mathbb{N}$. For every $i \in \{1\ldots n\}$, there exist a formula ϕ_i and valuation V_i' such that $\langle T',V_i'\rangle \in [\![\phi_i]\!]$ and $\langle U_i,V_i'\rangle \notin [\![\phi_i]\!]$ or there exists a variable v such that $V_i(v) \neq V''(v)$. Note that if there exists a v such that $V_i(v) \neq V''(v)$ then there is also a distinguishing formula ϕ_i: $(v = V''(v))$.

We are now in a position to construct a formula that is satisfied by $\langle T,V'\rangle$ but not by $\langle U,V'\rangle$, contradicting our assumption that $\langle T,V'\rangle$ and $\langle U,V'\rangle$ satisfy the same formulas. The formula

$$\langle\{\lambda\}\rangle(\downarrow(V_1')\phi_1 \wedge \downarrow(V_2')\phi_2 \wedge \cdots \wedge \downarrow(V_n')\phi_n)$$

is satisfied by $\langle T,V'\rangle$ but not by $\langle U,V'\rangle$. $\square$

We now have a logic with an operator to conveniently refer to the contents of global variables and which corresponds to the appropriate behavioural equivalences.

7.4 Verifying HML^{check} Formulas Using Selfloops

We would like to have tools to verify HML^{check} formulas for a given LTS in practice. Adjusting existing model-checking algorithms would be cumbersome. In this section we will show that we can check a formula by translating it to a plain HML formula and transforming the LTS by 'annotating' each state with *value* selfloops labelling them with the value each variable has. We first illustrate the intent with an example.

Example 7.14. *Consider the two following LTS fragments. The states in the top LTS fragment have an explicit valuation in the states. The states in the bottom LTS fragment have been obtained by transforming the top LTS fragment and do not have an explicit valuation, but do have value-labelled selfloops representing the valuation of the global variables in the original LTS. Suppose we want to verify the formula* $[a](v = 1)$ *for state* $\langle P,V\rangle$, *with* $V(v) = 0$. *This formula holds if and only if for state* s *the formula* $[a]\langle value(v,1)\rangle\top$ *holds.*

$$\begin{tikzcd}\langle P, V\rangle \arrow[r, "a"] & \langle \delta, V[v \mapsto 1]\rangle\end{tikzcd}$$

$$\begin{tikzcd}s \arrow[loop above, "value(v,0)"] \arrow[r, "a"] & t \arrow[loop above, "value(v,1)"]\end{tikzcd}$$

The selfloops labelled with *value* provide information on the values of global variables in every state, which we will exploit in the translation of HML^{check} formulas. Given a HML^{check} formula we replace each occurrence of the check operator of the shape $(v = e)$ with $\langle value(v, e)\rangle\top$, where $v \in \mathbb{V}$ and $d \in D$.

Definition 7.15. *Let θ denote the translation form HML^{check} formulas to HML, which we define inductively, where v and e range over $\mathbb{V}$ and D, respectively:*

$$\begin{array}{lll}\theta(\top) & = & \top, \\ \theta(v = e) & = & \langle value(v, e)\rangle\top, \\ \theta(\neg\phi) & = & \neg\theta(\phi), \\ \theta(\phi_1 \wedge \phi_2) & = & \theta(\phi_1) \wedge \theta(\phi_2), \\ \theta(\langle T\rangle\phi) & = & \langle T\rangle\theta(\phi).\end{array}$$

Let $\mathcal{L}_V$ denote the extended set of transition labels:

$$\mathcal{L}_V = \mathcal{L} \cup \{value(v, d) \mid v \in \mathbb{V}, d \in D\}$$

We proceed by defining a transformation Λ on the LTS.

Definition 7.16. *Let $(S_1 \times \mathcal{V}, \mathcal{L}, \rightarrow_1)$ be an LTS. Then $\Lambda((S_1 \times \mathcal{V}, \mathcal{L}, \rightarrow_1)) = ((S_2, \mathcal{L}_V, \rightarrow_2), F)$ where:*

1. *$F : (S_1 \times \mathcal{V}) \rightarrow S_2$ is a bijective function;*
2. *for all $v \in \mathbb{V}, d \in D$ we have that $s \xrightarrow{value(v,d)}_2 t$ if and only if $s = t$ and $F^{-1}(s) = \langle P, V\rangle$ such that $V(v) = d$;*
3. *and for all states $s, t \in S_2$ and labels $\lambda \in \mathcal{L}$ we have that $s \xrightarrow{\lambda}_2 t$ if and only if $F^{-1}(s) \xrightarrow{\lambda}_1 F^{-1}(t)$.*

Note that a bijective function $F : (S_1 \times \mathcal{V}) \rightarrow S_2$ always exists, the identity function suffices. In our definition it is purposely left opaque what the contents of S_2 is and how F is defined as they are irrelevant.

Theorem 7.17, below, is the culmination of this section. It states that transformed formulas hold for states in the transformed LTS if and only if the original formula holds in the corresponding state in the original LTS.

Theorem 7.17. *Let $(S_1 \times \mathcal{V}, \mathcal{L}, \rightarrow_1)$ be an LTS and let $\Lambda((S_1 \times \mathcal{V}, \mathcal{L}, \rightarrow_1)) = ((S_2, \mathcal{L}_V, \rightarrow_2), F)$. Let $\langle P, V\rangle \in S_1 \times \mathcal{V}$ and $t \in S_2$ and suppose $F(\langle P, V\rangle) = t$. For every HML^{check} formula ϕ we have that $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $t \in [\![\theta(\phi)]\!]$.*

Proof. We prove by structural induction on ϕ that $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $t \in [\![\theta(\phi)]\!]$ with the induction hypothesis that for any pair of states $s \in S_1 \times \mathcal{V}$ and $u \in S_2$ such that $F(s) = u$, any subformula ϕ' of ϕ holds for s if and only if $\theta(\phi')$ holds for u.
Base cases

- $\phi = \top$: $[\![\top]\!] = S_1 \times \mathcal{V}$ so $\langle P, V\rangle \in [\![\top]\!]$. $[\![\theta(\top)]\!] = [\![\top]\!] = S_2$ so $t \in [\![\theta(\top)]\!]$.
- $\phi = (v = e)$: $[\![v = e]\!] = \{\langle U, V'\rangle \in S_1 \times \mathcal{V} \mid V'(v) = e\}$ so $\langle P, V\rangle$ is in $[\![v = e]\!]$ if and only if $V(v) = e$. Since $F(\langle P, V\rangle) = t$ we can derive that $t \xrightarrow{value(v,e)}_2 t$ if and only if $V(v) = e$. $\theta((v = e)) = \langle value(v, e)\rangle\top$ so we conclude that $t \in [\![\theta(\phi)]\!]$ if and only if $V(v) = e$.

Step

- $\phi = \neg\phi'$: $[\![\neg\phi']\!] = S_1 \times \mathcal{V} \setminus [\![\phi']\!]$ so $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle P, V\rangle \notin [\![\phi']\!]$. By the induction hypothesis $t \in [\![\theta(\phi')]\!]$ if and only if $\langle P, V\rangle \in [\![\phi']\!]$. $\theta(\neg\phi') = S_2 \setminus [\![\phi']\!]$. Hence, $t \in [\![\theta(\phi)]\!]$ if and only if $\langle P, V\rangle \in [\![\phi]\!]$.

- $\phi = \phi' \wedge \phi''$: $[\![\phi' \wedge \phi'']\!] = [\![\phi']\!] \cap [\![\phi'']\!]$. $\theta(\phi' \wedge \phi'') = \theta(\phi') \wedge \theta(\phi'')$, so $[\![\theta(\phi' \wedge \phi'')]\!] = [\![\theta(\phi')]\!] \cap [\![\theta(\phi'')]\!]$. By the induction hypothesis $\langle P, V\rangle \in [\![\phi']\!]$ if and only if $t \in [\![\theta(\phi')]\!]$ and $\langle P, V\rangle \in [\![\phi'']\!]$ if and only if $t \in [\![\theta(\phi'')]\!]$. Hence $\langle P, V\rangle \in [\![\phi' \wedge \phi'']\!]$ if and only if $t \in [\![\theta(\phi' \wedge \phi'')]\!]$.
- $\phi = \langle T\rangle\phi'$: $[\![\langle T\rangle\phi']\!] = \{u \in S_1 \times \mathcal{V} \mid \exists u' \in [\![\phi']\!], \lambda \in T : u \xrightarrow{\lambda}_1 u'\}$. $\theta(\langle T\rangle\phi') = \langle T\rangle\theta(\phi')$ so $[\![\theta(\langle T\rangle\phi')]\!] = \{u \in S_2 \mid \exists u' \in [\![\phi']\!], \lambda \in T : u \xrightarrow{\lambda}_2 u'\}$. We have that $\langle P, V\rangle \in [\![\phi]\!]$ if and only if there exists a state u and a label $\lambda \in T$ such that $\langle P, V\rangle \xrightarrow{\lambda}_1 u$ and $u \in [\![\phi']\!]$. Since $F(\langle P, V\rangle) = t$ we know that there exists some state t' such that $F(u) = t'$ and $t \xrightarrow{\lambda}_2 t'$ if and only if $\langle P, V\rangle \xrightarrow{\lambda}_1 u$. By the induction hypothesis $t' \in [\![\theta(\phi')]\!]$ if and only if $u \in [\![\phi']\!]$. By the semantics of the diamond operator we conclude that $t \in [\![\theta(\phi)]\!]$ if and only if $\langle P, V\rangle \in [\![\phi]\!]$.

□

7.5 Relation to FormaSig

FormaSig models do not truly have global data or even shared data. All the data is local to a specific state machine. Components cannot observe or act upon the data of other components. The common ground is that we want states to contain data that we can inspect whilst model checking. We are interested in the state of a component: the SysML state the component is in, the contents of the communication buffer and the valuation of local variables.

The attentive reader might have noticed that the *value*-labelled selfloops of the preceding sections are very similar to some of the selfloops that are used in FormaSig models: `inState`, `varVal` and `inEventPool`. They also serve a very similar purpose; with the formula $\langle \texttt{inState}(c, s)\rangle\top$ we can check whether component c is in state s.

The logic used by mCRL2, an extension of the modal μ-calculus, demands some extra carefulness when introducing selfloops. It allows us to use action formulas in the box and diamond operator, e.g. in the formula $[a.\top]\bot$, $\top$ matches any multi-action so the formula expresses that after any a-labelled transition we end up in a state without outgoing transitions. The formula might no longer hold after we introduce selfloops. To repair the formula we need to exclude the selfloops. Assuming there are only `inState` selfloops we can repair the formula as follows.

$$[a.\neg(\exists_{c:\texttt{CompName},s:\texttt{StateName}}\ \texttt{inState}(c, s))]\bot$$

The second action formula will now match any multi-action that is not the singleton multi-action `inState`.

7.6 Scoped Variables

So far we have considered variables that are global, and moreover every variable v is accessible globally and exists in every state. Suppose we want to have more fine grained control by sharing a variable only between specific parallel components.

We begin by discussing important earlier work on this subject. In [1] propositional signals and a state operator are presented. A state operator tracks the state of something, say the current colour of a traffic light. To update the state, a mapping *effect* is used that, given an action and the current state, produces the next state. By example, if the traffic light is red and the action *change* should turn it to green then $\mathit{effect}(\mathit{change}, \mathit{red}) = \mathit{green}$. To make the current state known to parallel processes, propositional signals are used. A mapping *sig* associates a set of propositions to the current state. Sticking to the example of the traffic light we could define $\mathit{sig}(\mathit{red}) = r \wedge \neg g$. These signals can be picked up by parallel processes using a guard, for example $(g) \to \mathit{drive}$. By default the propositional signals are global, but a *signal hiding* operator can be used to create scoping.

The theory is elegant and flexible but in practise we would not want to specify the states, and *effect* function. It would even be impossible to specify explicitly when we would use the state operator to model a variable with an infinite data domain. For our simple process language with only comparisons in guards it would be rather straightforward to generate the states of the state operator and the functions *effect* and *sig*. There would be a state for every element in the data domain. For every action label *effect* is the identity function and for an assign action α we update the state according to the value being set. We could introduce a propositional signal for every element of the data domain and let *sig* ensure that the signal corresponding to the current value is emitted.

Keeping in mind that the end goal is integrating shared (or global) variables in mCRL2 we should also evaluate whether the state operator is suitable in that setting. In mCRL2 a wide range of Boolean expressions is allowed as a guard, so we should also be able to reference multiple shared variables. Suppose for example that we have a guard $x+y = 10$ where x and y are integer-valued shared variables. Let us also suppose that the variables have a different scope, i.e. they should be specified with distinct state operators. Both state operators could communicate their value by means of a propositional signal, e.g. the signals $x6$ and $y4$ communicating $x = 6$ and $y = 4$, for which the guard holds. There are, however, infinitely many combinations for x and y for which the guard evaluates to true so even this simple guard cannot be finitely specified.

Moreover, in mCRL2, actions can have data parameters. We should also be able to use shared variables in these action parameters. Similarly, we should be able to use shared variables in recursion. Complex expressions on the data should be possible. As an example, suppose there is an integer-valued shared variable p; then the following process definition expresses that if a local variable *myPos* is smaller than p then the process *Player* can jump to position p indicating with an action parameter how far he jumped.

$$Player(myPos : Int) = (myPos < p) \rightarrow jump(p - myPos).Player(p);$$

Based on these observations we propose an alternative way of modelling shared variables. We extend the syntax as presented in Section 7.1 with a new operator: the shared variable operator $G_{x,v}$, where $x \in \mathbb{V}$ and $v \in D$. The shared variable operator in $G_{x,v}(P)$ declares a variable x in the process P. When $P \xrightarrow{\alpha(x,v')} P'$ the change of valuation updates the shared variable operator: $G_{x,v}(P) \xrightarrow{\alpha(x,v')} G_{x,v'}(P')$.

To define how a variable is substituted with its value we introduce an auxiliary substitution operator S_V, where $V \in \mathcal{V}$. A transition $S_V((v = e) \rightarrow P) \xrightarrow{\lambda} P'$ is only possible when $V(v) = e$ (and $P \xrightarrow{\lambda} P'$). Let the set of process expressions generated by the extended grammar be denoted by $\mathcal{P}_s$.

We now want to associate an LTS with the process algebra. Valuation functions are now partial functions $\mathcal{V} : \mathbb{V} \hookrightarrow D$. Let $\epsilon \in \mathcal{V}$ denote the valuation function that is undefined for every variable. Let $V|V'$ denote the valuation function for which for all $v \in \mathbb{V}$, $V|V'(v) = V'(v)$ if $V'(v)$ is defined and $V|V'(v) = V(v)$ otherwise. The set of states of the LTS is $\mathcal{P}_s$. The transition relation is the least relation on states satisfying the rules of the structural operational semantics (see Table 7.1). The SOS rules are in path format so bisimulation is a congruence on the algebra.

Example 7.18. *Let us revisit the example of a car and a traffic light controller. We first define the recursive specification:*

$$\begin{aligned}
Car &:= (t=g)\to a.Car + (t=r)\to b.Car\\
TLC &:= (t=g)\to \alpha(t,r).TLC + (t=r)\to \alpha(t,g).TLC
\end{aligned}$$

The process expression that captures the interaction between the car and the traffic light controller is the following: $G_{t,r}(Car \parallel TLC)$.

The following derivation shows how the shared variable operator is updated when the traffic light turns green.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{S_{\epsilon[t\to r]}(\alpha(t,g).TLC) \xrightarrow{\alpha(t,g)} TLC} \qquad \dfrac{}{S_{\epsilon[t\to r]}(t)=r}}{S_{\epsilon[t\to r]}((t=r)\to\alpha(t,g).TLC) \xrightarrow{\alpha(t,g)} TLC}}{\begin{array}{c}S_{\epsilon[t\to r]}((t=g)\to\alpha(t,r).TLC\\ +(t=r)\to\alpha(t,g).TLC)\end{array} \xrightarrow{\alpha(t,g)} TLC}}{S_{\epsilon[t\to r]}(TLC) \xrightarrow{\alpha(t,g)} TLC}}{S_{\epsilon[t\to r]}(Car \parallel TLC) \xrightarrow{\alpha(t,g)} Car \parallel TLC}}{G_{t,r}(Car \parallel TLC) \xrightarrow{\alpha(t,g)} G_{t,g}(Car \parallel TLC)}$$

Conditionals referencing variables that are not declared are always evaluated negatively. Transitions labelled with variable assignments outside the scope of a matching G operator can be fired but will not update anything. In most settings it does not make sense to use variables that are out of scope so we may want to restrict ourselves to process expressions where every reference to a variable is in the scope of a G operator.

The set of free variables of a process expression P is denoted by $\mathsf{FV}(P)$ and is defined inductively below.

$$\begin{array}{lcl}
\mathsf{FV}(a.P) & = & \mathsf{FV}(P)\\
\mathsf{FV}(\alpha(v,d).P) & = & \{v\}\cup\mathsf{FV}(P)\\
\mathsf{FV}(\delta) & = & \emptyset\\
\mathsf{FV}(P+Q) & = & \mathsf{FV}(P)\cup\mathsf{FV}(Q)\\
\mathsf{FV}(P\parallel Q) & = & \mathsf{FV}(P)\cup\mathsf{FV}(Q)\\
\mathsf{FV}(\partial_B(P)) & = & \mathsf{FV}(P)\\
\mathsf{FV}((v=e)\to P) & = & \{v\}\cup\mathsf{FV}(P)\\
\mathsf{FV}(X) & = & \mathsf{FV}(P),\text{ if } X\overset{\text{def}}{=}P\\
\mathsf{FV}(S_V(P)) & = & \mathsf{FV}(P)\\
\mathsf{FV}(G_{v,d}(P)) & = & \mathsf{FV}(P)\setminus\{v\}
\end{array}$$

A process expression P is *closed* if and only if $\mathsf{FV}(P)=\emptyset$. Note that we do not consider the process expression $S_V(\alpha(v,d).P)$ to be closed; it is only closed when variables are in the scope of a shared variable operator. The lemma below states that any process expression reachable from a process expression that is closed is also closed. Hence, a static analysis can be done on the specification guaranteeing that in every state of the state space all shared variables are in scope. It is a mild syntactic restriction.

Lemma 7.19. *For any closed process expression $P \in \mathcal{P}_s$ all reachable process expressions $P' \in \mathcal{P}_s$, $P \to^* P'$, are also closed.*

Proof. From the definition of FV and the conclusions of the SOS rules it follows that for any transition $P_1 \xrightarrow{\lambda} P_2$, it is the case that $\mathsf{FV}(P_2) \subseteq \mathsf{FV}(P_1)$ so we also have that P_2 is closed if P_1 is closed. The statement in the lemma follows from straightforward induction on the length of the path from P to P'. □

A significant benefit of shared variables over global variables is that it is purely process based and event based; we can abstract from the contents of a state, all the information is embedded in the process expression and strong bisimulation is a congruence of the process algebra. In the process algebra with propositional signals this is not the case; the signals are emitted globally by default, and a state consists of a process expression and a valuation. A significant downside is directly related to the benefit: we cannot inspect the contents of a state during model checking. We might add selfloops like in Section 7.4 but we cannot do that in a generic way since a variable may be declared multiple times.

7.7 Related Work

In the early days of process algebra, doing away with global variables in favour of message passing and local variables was an important paradigm shift [2]. Since then there have, nevertheless, been some efforts to reintroduce notions of globally available data.

In the previous section we have already discussed propositional signals and the state operator. Other approaches, such as the one presented in [109, Chapter 19], model global variables as separate parallel processes and use a protocol to ensure only one process accesses a global variable at a time. This approach introduces extra internal steps, which increases the state space. Moreover, it introduces divergence when a process locks a global variable, reads the value, concludes that it cannot make a step and unlocks the variable again.

Formalisms based on Concurrent Constraint Programming (CCP) [110] have global data at their core. In CCP a central store houses a set of constraints. Concurrent processes may *tell* a constraint, adding it to the global store or *ask* a constraint, checking whether it is entailed by the constraints in the store. An ask will block until other processes have added sufficient constraints to the store. Process calculi based on the coordination language LINDA [93] also use global data. In these process calculi there is a global set of data elements. Similarly to CCP, processes may tell a data element (adding it to the global set) or ask a data element (checking whether it is in the set). Additionally, processes may *get* an element, removing it from the data set. LINDA does not have a concept of variables, just a central set of data elements. To the best of our knowledge process calculi based on CCP or LINDA do not allow asking a constraint/data element and acting upon the information in a single step.

7.8 Conclusion

In this chapter we have presented a simple process calculus with global variables and studied various aspects of it. To start we examined appropriate notions of equivalence: stateless bisimulation for process expressions and state-based bisimulation for states. Then, for our first contribution we presented a logic extending HML with a check and a set operator and proved that HML^{check} is strong enough to differentiate states that are not state-based bisimilar and HML$^{check+set}$ is strong enough to differentiate process expressions that are not stateless bisimilar. As a second contribution we showed a way to verify requirements referencing data variables using traditional tools by adding selfloops. Finally, we explored an alternative approach with shared variables.

$\dfrac{}{\lambda.P \xrightarrow{\lambda} P}$	$\dfrac{}{S_V(\lambda.P) \xrightarrow{\lambda} P}$
$\dfrac{P \xrightarrow{\lambda} P'}{\partial_B(P) \xrightarrow{\lambda} \partial_B(P')}\,\lambda \notin B$	$\dfrac{S_V(P) \xrightarrow{a} P'}{S_V(\partial_B(P)) \xrightarrow{a} \partial_B(P')}\,a \notin B$
$\dfrac{P \xrightarrow{\lambda} P'}{P + Q \xrightarrow{\lambda} P'}$	$\dfrac{S_V(P) \xrightarrow{\lambda} P'}{S_V(P + Q) \xrightarrow{\lambda} P'}$
$\dfrac{Q \xrightarrow{\lambda} Q'}{P + Q \xrightarrow{\lambda} Q'}$	$\dfrac{S_V(Q) \xrightarrow{\lambda} Q'}{S_V(P + Q) \xrightarrow{\lambda} Q'}$
$\dfrac{P \xrightarrow{\lambda} P'}{P \parallel Q \xrightarrow{\lambda} P' \parallel Q}$	$\dfrac{S_V(P) \xrightarrow{\lambda} P'}{S_V(P \parallel Q) \xrightarrow{\lambda} P' \parallel Q}$
$\dfrac{Q \xrightarrow{\lambda} Q'}{P \parallel Q \xrightarrow{\lambda} P \parallel Q'}$	$\dfrac{S_V(Q) \xrightarrow{\lambda} Q'}{S_V(P \parallel Q) \xrightarrow{\lambda} P \parallel Q'}$
$\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P \parallel Q \xrightarrow{c} P' \parallel Q'}\,\gamma(a,b) = c$	$\dfrac{S_V(P) \xrightarrow{a} P' \quad S_V(Q) \xrightarrow{b} Q'}{S_V(P \parallel Q) \xrightarrow{c} P' \parallel Q'}\,\gamma(a,b) = c$
$\dfrac{P \xrightarrow{\lambda} P'}{X \xrightarrow{\lambda} P'}\,X \stackrel{\text{def}}{=} P$	$\dfrac{S_V(P) \xrightarrow{\lambda} P'}{S_V(X) \xrightarrow{\lambda} P'}\,X \stackrel{\text{def}}{=} P$
$\dfrac{S_{V[x \to v]}(P) \xrightarrow{a} P'}{S_V(G_{x,v}(P)) \xrightarrow{a} G_{x,v}(P')}$	$\dfrac{S_{V[x \to v]}(P) \xrightarrow{\alpha(x,v')} P'}{S_V(G_{x,v}(P)) \xrightarrow{\alpha(x,v')} G_{x,v'}(P')}$
$\dfrac{S_V(P) \xrightarrow{\lambda} P' \quad V(v) = e}{S_V((v = e) \to P) \xrightarrow{\lambda} P'}$	$\dfrac{S_{V \mid V'}(P) \xrightarrow{a} P'}{S_V(S_{V'}(P)) \xrightarrow{a} P'}$
$\dfrac{S_{\epsilon[x \to v]}(P) \xrightarrow{a} P'}{G_{x,v}(P) \xrightarrow{a} G_{x,v}(P')}$	$\dfrac{S_{\epsilon[x \to v]}(P) \xrightarrow{\alpha(x,v')} P'}{G_{x,v}(P) \xrightarrow{\alpha(x,v')} G_{x,v'}(P')}$

Table 7.1: Structural operational semantics for $\mathcal{P}_s$ expressions.

Chapter 8

Conclusions and Future Work

Life is not a problem to be solved,
but a reality to be experienced.

Søren Kierkegaard

The main goal of the FormaSig project was to enable formal verification and model-based testing of EULYNX models (see Figure 1.1). The main research question addressed in this thesis, as stated in Chapter 1 is how can we formalise the semantics of EULYNX SysML and effectively verify requirements for industrial models?

A partially automated translation has been achieved from the EULYNX SysML dialect to mCRL2, giving a formal interpretation to the original models. The process of formalisation uncovered some ambiguities, as was expected. We have reported them to the Modelling and Testing cluster of EULYNX. Our formalisation resolves these ambiguities.

To be able to handle industrial (EULYNX) models, improvements in the scalability of verification were needed. The techniques discussed in Chapter 4 have been developed over the course of several years. Together they brought a dramatic improvement in performance. We are now able to handle most EULYNX models in contrast to a few years ago.

For a number of interfaces we have also achieved the goal of formal verification. In collaboration with ProRail and DB Netz AG requirements have been formulated, which in turn have been formalised as modal μ-calculus formulas and verified using the mCRL2 toolkit. These verification efforts have uncovered multiple errors and missing assumptions, which have been communicated to ProRail and DB Netz AG.

SysML is state-based whereas mCRL2 is action-based. This gave some challenges in formulating requirements as we need to refer to the SysML states. This issue was overcome by adding `inState` selfloops. In Chapter 7 we explored ways to integrate the state-based and action-based worlds in a comprehensive manner.

Another challenge arose when verifying liveness requirements. The LTS permits infinite runs of the system where one or more components never get the turn to take a transition. Such runs may violate a liveness requirement but are not realistic. By adding a justness assumption we were able to rule out unrealistic liveness violating paths without resorting to assumptions like fairness, which themselves may not be realistic.

The research on justness and global variables was not linked to FormaSig initially. The work on justness even started in a project some time before the start of my PhD. This demonstrates the interconnectedness of research. Even in my own research one project was, unexpectedly, directly applicable in another. Hopefully, the contents of this thesis will also prove useful to others.

In the remainder of this chapter we reflect on the state of formal methods in the railway domain (Section 8.1). We also suggest a number of ideas for future research directions to improve the tools discussed in this thesis (Section 8.2), and in general (Section 8.3).

8.1 Formal Methods in the Railway Domain

In FormaSig, as with many other applications of formal methods in the railway domain, there is a cooperation between railway infrastructure managers and academic institutions. The former possess the domain knowledge but not the mathematical knowledge and, conversely, the academic institutions possess the expertise on formal methods but lack domain knowledge. As a result, the infrastructure managers are happy with the contributions of formal methods in finding flaws but do not fully understand the formal models and are not able to modify them or assess their quality. Since they are responsible for the quality and safety of signalling systems and need to be able to assess the quality of specifications (in internal reviews) they are reluctant to make a formal model the authoritative specification. Instead, a semi-formal SysML model or natural language specification is authoritative. Even SysML is considered somewhat problematic as not every signalling engineer is able to fully comprehend the models, especially the state machines. For this reason some infrastructure managers translate the EULYNX SysML models to natural language for their national requirements in tendering processes.

In FormaSig, formal methods are applied to an existing specification, as is more often the case in collaborations between infrastructure managers and academia. The existing models are, however, not optimised for verification. EULYNX models break down functionality into many different (parallel) components, leading to a huge state space. We conjecture that if these models were created with the goal of formal verification in mind the state space would be (much) smaller. Ideally, modelling in SysML and verifying with mCRL2 would go hand in hand. Working in tandem also creates a form of redundancy, lowering the chance of defects in the final product [22].

We conclude that formal methods are, despite their long history, not yet standard practice in the railway signalling domain. This might be improved if rail infrastructure managers would obtain in-house knowledge of formal methods, just like they have in-house electrical engineering experts. Then they would have people who can judge the quality of a formal analysis and who can incorporate formal methods during the inception of new projects.

We do believe formal methods could be beneficial w.r.t. the challenges faced by the railway infrastructure managers. The complexity of systems is increased by switching from instant communication over copper wiring to packet based asynchronous communication. Formal methods can help uncover issues arising from this, as demonstrated in this thesis.

It can also help in contracting suppliers. By creating specifications using a formal language (many) ambiguities are eliminated, which should speed up implementation as less back and forth discussions are needed. Additionally, model checking ensures that the specification delivered to the supplier is correct. The chance of finding ambiguities or flaws in a late stage is therefore reduced. This is extra important for (European) standards. Ambiguities and flaws may be resolved slightly differently in different projects and require patches to the standard, creating a patchwork of implementations, which hinders interoperability. Model-based testing could also help to test more thoroughly and thereby catch potential problems still before deployment.

The challenges faced by the railway domain in designing correct protocols are similar to other domains. The software and protocols used in, for example, aeroplanes and in the oil and gas industry are also safety critical. Our knowledge of these other domains is too limited to make a thorough comparison. We conjecture that high-tech companies such as Intel [41], Amazon [92] and Facebook [32] are relatively quick at adopting formal methods. They have the benefit of having less governmental oversight dictating in what way correctness should be established and are therefore more flexible to adopt new technologies. Furthermore, they are financially extremely motivated to have correct software and have a large financial capacity to explore new ways to improve their products. An example of a public organisation that has a lot of experience with formal methods is NASA. They have the Langley Formal Methods Research Program and organise an annual symposium on the topic: the NASA Formal Methods Symposium (NFM).

8.2 Ideas for Future Extensions of FormaSig Tools

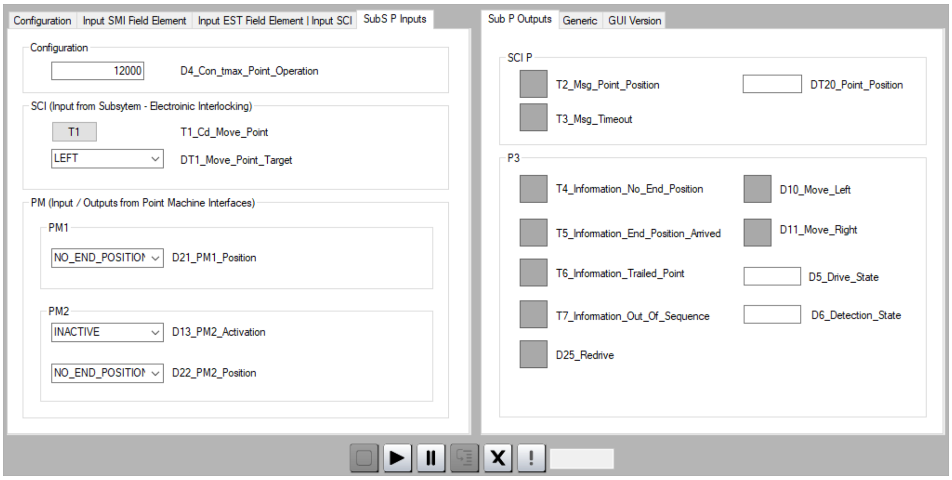
Over the years many ideas have popped up in the context of FormaSig to make verification of EULYNX SysML models more convenient. Unfortunately there is not enough time in a PhD project to pursue every idea. Below a list is given of unexplored ideas.

Visualise Counterexamples. Evidence provided by the mCRL2 toolkit is presented as an LTS with labels from the mCRL2 model. In the future we would like to improve usability by converting these evidence LTSs to UML sequence diagrams. This may not always be possible (or beneficial) as the evidence LTS may contain the entire state space. We suspect that some common evidence structures such as simple traces and lassos (loops, possibly with a trace leading up to it) are well suited for conversion to sequence diagrams.

Simple Requirements Language. At the moment, formulating formal requirements is a somewhat cumbersome process. To verify requirements, they need to be formulated as μ-calculus formulas. These formulas are not readable for signalling engineers. Identifying and formulating requirements is therefore a time consuming process involving back and forth translation between natural language and μ-calculus formulas. There is a substantial body of research in creating formal requirements languages that are more intuitive. Some requirements languages are based on structured text, i.e. requirements are sentences built out of specific keywords, where each keyword has a formal interpretation. For example, the requirement "Whenever component X is in state Y, then always eventually Component A will be in state B", might be a requirement where "whenever ... then", "always eventually", etcetera have a formal interpretation. There are also visual requirements languages, such as live sequence charts [23, 27], which capture requirements in a format close to UML sequence diagrams. The visual/textual requirements language should be translated automatically to μ-calculus formulas. We think that having such an intuitive requirements language would benefit the communication between formal methods experts and signalling engineers.

Simulator Based on the mCRL2 Model. In the EULYNX project, software simulators of the interface are derived semi-automatically using tools provided by the PTC modelling toolkit. Most signalling engineers are not aware of (the subtleties of) the semantics of SysML models. These simulators let signalling engineers play around with the behaviour defined in the model, validating that it is in accordance with their expectations (see Figure 8.1). The semantics PTC gives to these simulators is not properly defined and might deviate from the formal semantics of our mCRL2 model. Ideally, the simulator would behave exactly the same as the more accurate mCRL2 model. A solution might be to built a simulator on top of the mCRL2 model. It would be even better if the UML/SysML community decides on a formal semantics of the language which should then be implemented by the vendors of SysML tools, such as PTC and our toolchain.

Fully Automatic Formalisation. At the moment the SysML models need to be manually converted to jEULYNX, which is cumbersome and error prone. It would be better to load the SysML models directly, which is hindered by the fact that the PTC tool used by EULYNX does not allow for XMI exports that are compliant with the OMG standard. An XMI to XMI conversion tool could be developed. Once SysML v2 is introduced and adopted by PTC loading the models may become easier as SysML v2 introduces a textual format besides the graphical diagram format [96].



Configuration

12000 D4_Con_tmax_Point_Operation

SCI (Input from Subsytem - Electroinic Interlocking)

T1 T1_Cd_Move_Point

LEFT DT1_Move_Point_Target

PM (Input / Outputs from Point Machine Interfaces)

PM1

NO_END_POSITION D21_PM1_Position

PM2

INACTIVE D13_PM2_Activation

NO_END_POSITION D22_PM2_Position

SCI P

T2_Msg_Point_Position DT20_Point_Position

T3_Msg_Timeout

P3

T4_Information_No_End_Position D10_Move_Left

T5_Information_End_Position_Arrived D11_Move_Right

T6_Information_Trailed_Point D5_Drive_State

T7_Information_Out_Of_Sequence D6_Detection_State

D25_Redrive

Figure 8.1: A view of the GUI of the Point simulator. Inputs are located on the left, outputs on the right, and simulation controls at the bottom.

8.3 Miscellaneous Ideas for Future Research Directions

Research typically both answers questions and uncovers new questions and challenges. Below several ideas for (research) directions are listed.

Integration of Global Variables in Existing Tools. Possible future work is to integrate the concept of global variables in existing more advanced process algebras and their associated toolsets, for several reasons. Building upon existing algebras makes it accessible to a wide range of users and unlocks tool support. Moreover, process algebras supporting complex data operations in guards could more fully exploit the benefits of global variables. It would be interesting to incorporate global variables in the mCRL2 toolset.

A challenge in incorporating global variables in mCRL2 is formed by the concept of multi-actions. In particular, a solution needs to be found for colliding assignments: what is the value of a variable when a transition labelled with a multi-action consisting of two assignment actions $\alpha(v, d)|\alpha(v, e)$ is performed? We would probably want to prevent such transitions from happening, for which we see two possible solutions. The first option is to prevent it at the semantic level by disallowing the hiding or communication of assignment actions and adapting the SOS rules to rule out colliding assignments. Another option is to enforce restrictions on the syntactic level, to only define a semantic interpretation to some subset of well-formed process expressions. For example we could enforce that for each variable only one parallel component can change it; for every state P reachable from the initial state, every variable $v \in \mathbb{V}$ and every sub-expression of the shape

$Q \parallel R$ in P we have that $Q \rightarrow^* Q' \xrightarrow{\alpha(v,d)} Q''$ or $R \rightarrow^* R' \xrightarrow{\alpha(v,d)} R''$, but not both.

Debugging Rewrite Specifications in mCRL2. In the formalisation of state machine behaviour we have made extensive use of the data language offered by mCRL2. These data equations are used as rewrite rules by the tools to rewrite data expressions to a normal form (e.g., by rewriting guards in an LPS to true or false). Such a rewrite specification is in itself a 'software' artefact that may contain bugs.

The parser catches bugs related to mismatching types. Other, (subtle) mistakes in data equations can be hard to debug. We encountered the following two classes of bugs.

1. A mapping may not produce the intended output for every input. In some cases this will lead to an error whilst exploring the state space, e.g. because a guard does not evaluate to true or false. In other cases there is no error but the LTS is not as intended. When the LTS is large and the bug only occurs in specific corner cases this is hard to discover.

2. The rewrite specification may not terminate, e.g. because there exists a loop in the rewrite path. This manifests itself as a stall or crash whilst linearising or exploring the state space (without giving any feedback to the user).

Common debugging techniques from programming could be used in this setting. Unit tests and breakpoints would be beneficial in finding the first class of bugs. In general it is undecidable to check whether a rewrite specification is terminating, but through static analysis some cases of non-termination might be detected. It would also be helpful to be able to step through the term rewriting path; ideally with a cycle detection tool. The rewrite engine is highly optimised so modifying it with debugging tools is not desirable; a second unoptimised rewrite engine with debugging tools would make more sense, analogous to how many compilers allow optimised compilation as well as unoptimised compilation with debug symbols.

An Authoritative Semantic Interpretation of SysML. Semi-formality is both a strength and a weakness of SysML. It allows designing an architecture without committing to details. On the other hand, verifying whether a design meets the requirements is hampered. As discussed in Chapter 3 there have been many scientific papers on the subject of formalising and verifying SysML models. Each paper focusses on a specific subset of diagrams and a specific action language and has its own formalisation approach.

It would be desirable to formalise a large subset of SysML without committing to implementation details. A way to achieve this is to formally capture what degrees of freedom there are, as we have done with the action language in our formalisation. You can mathematically specify that there is some set of action language expressions and some mapping to evaluate them without committing to a specific action language. Many variation points in SysML could be specified in such a way, e.g. the event pool. Common options (such as a FIFO queue for the event pool, and OCL for the action language) could also be formally specified.

The different diagram types offer different viewpoints on a system and should also be formalised in a different way. Behavioural diagrams such as State Machines and Activity Diagrams could be given a semantic interpretation in terms of a (timed) LTS. For structural diagrams (such as IBDs and Block Definition Diagrams) it should be formalised how a set of diagrams can be combined into a comprehensive system overview. The semantic interpretation of such a system overview combined with the behavioural diagrams defining the behaviour of components could again be expressed in terms of a (timed) LTS. These definitions could again be parametrised giving multiple options for the concurrency model. For sequence diagrams it could be formalised when its behaviour is included in the system (as defined by the structural and behavioural diagrams).

Such a formalisation of SysML is a massive effort that cannot be undertaken by a small group of researchers. A large collaborative effort of multiple universities (and companies) would be needed. Ideally it would be done in close collaboration with the OMG group, resulting in an official authoritative document, as is the case with PSSM.

The same approach could be applied to UML, which shares many of its diagram types with SysML.

Bibliography

Beware you be not swallowed up in books! An ounce of love is worth a pound of knowledge.

John Wesley

[1] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2009. DOI: `10.1017/CBO9781139195003` (cit. on pp. 160, 172).

[2] J.C.M. Baeten. "A brief history of process algebra". In: *Theoretical Computer Science* 335.2-3 (May 2005), pp. 131–146. DOI: `10.1016/j.tcs.2004.07.036` (cit. on p. 175).

[3] Maarten Bartholomeus, Bas Luttik, and Tim A. C. Willemse. "Modelling and Analysing ERTMS Hybrid Level 3 with the mCRL2 Toolset". In: *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings.* Ed. by Falk Howar and Jiri Barnat. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 98–114. DOI: `10.1007/978-3-030-00244-2_7` (cit. on p. 3).

[4] Davide Basile, Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, Franco Mazzanti, Andrea Piattino, Daniele Trentini, and Alessio Ferrari. "On the Industrial Uptake of Formal Methods in the Railway Domain - A Survey with Stakeholders". In: *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings.* Ed. by Carlo A. Furia and Kirsten Winter. Vol. 11023. Lecture Notes in Computer Science. Springer, 2018, pp. 20–29. DOI: `10.1007/978-3-319-98938-9_2` (cit. on p. 3).

[5] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. "Developing UPPAAL over 15 years". In: *Softw. Pract. Exp.* 41.2 (2011), pp. 133–142. DOI: 10.1002/spe.1006 (cit. on p. 24).

[6] Jan A. Bergstra and Jan Willem Klop. "Algebra of Communicating Processes with Abstraction". In: *Theor. Comput. Sci.* 37 (1985), pp. 77–121. DOI: 10.1016/0304-3975(85)90088-X (cit. on pp. 11, 117, 120).

[7] Bruno Blanchet. "Automatic verification of correspondences for security protocols". In: *J. Comput. Secur.* 17.4 (2009), pp. 363–434. DOI: 10.3233/JCS-2009-0339 (cit. on p. 24).

[8] Stefan Blom, Wan J. Fokkink, Jan Friso Groote, Izak van Langevelde, Bert Lisser, and Jaco van de Pol. "μCRL: A Toolset for Analysing Algebraic Specifications". In: *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings.* Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Vol. 2102. Lecture Notes in Computer Science. Springer, 2001, pp. 250–254. DOI: 10.1007/3-540-44585-4_23 (cit. on p. 3).

[9] Andrea Bonacchi, Alessandro Fantechi, Stefano Bacherini, and Matteo Tempestini. "Validation process for railway interlocking systems". In: *Sci. Comput. Program.* 128 (2016), pp. 2–21. DOI: 10.1016/j.scico.2016.04.004 (cit. on p. 3).

[10] Mark Bouwman. *Liveness analysis in process algebra: simpler techniques to model mutex algorithms.* Tech. rep. Available at http://www.win.tue.nl/~timw/downloads/bouwman_seminar.pdf. Eindhoven University of Technology, 2018 (cit. on p. 8).

[11] Mark Bouwman. "A model-based test platform for rail signalling systems". Master's thesis. Eindhoven University of Technology, 2019 (cit. on p. 207).

[12] Mark Bouwman. *Replication package for the PhD thesis "Supporting Railway Standardisation with Formal Verification".* 2023. DOI: 10.5281/zenodo.7852535 (cit. on pp. 6, 25, 30, 44, 45, 65, 71, 73, 76, 78, 150, 151).

[13] Mark Bouwman and Rick Erkens. "Term Rewriting Based On Set Automaton Matching". In: *CoRR* abs/2202.08687 (2022). arXiv: 2202.08687 (cit. on p. 207).

[14] Mark Bouwman, Bob Janssen, and Bas Luttik. "Formal Modelling and Verification of an Interlocking Using mCRL2". In: *Formal Methods for Industrial Critical Systems - 24th International Conference, FMICS 2019, Amsterdam, The Netherlands, August 30-31, 2019, Proceedings.* Ed. by Kim Guldstrand Larsen and Tim A. C. Willemse. Vol. 11687. Lecture Notes in Computer Science. Springer, 2019, pp. 22–39. DOI: 10.1007/978-3-030-27008-7_2 (cit. on pp. 3, 206).

[15] Mark Bouwman, Maurice Laveaux, Bas Luttik, and Tim A. C. Willemse. “Decompositional Branching Bisimulation Minimisation of Monolithic Processes”. In: *Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings*. Ed. by Silvia Lizeth Tapia Tarifa and José Proença. Vol. 13712. Lecture Notes in Computer Science. Springer, 2022, pp. 161–182. DOI: `10.1007/978-3-031-20872-0\_10` (cit. on pp. 7, 207).

[16] Mark Bouwman, Bas Luttik, Wouter Schols, and Tim A. C. Willemse. “A process algebra with global variables”. In: *Proceedings Combined 27th International Workshop on Expressiveness in Concurrency and 17th Workshop on Structural Operational Semantics, EXPRESS/SOS 2020, and 17th Workshop on Structural Operational Semantics, 31 August 2020*. Ed. by Ornela Dardha and Jurriaan Rot. Vol. 322. EPTCS. 2020, pp. 33–50. DOI: `10.4204/EPTCS.322.5` (cit. on pp. 9, 206).

[17] Mark Bouwman, Bas Luttik, and Djurre van der Wal. “A Formalisation of SysML State Machines in mCRL2”. In: *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*. Ed. by Kirstin Peters and Tim A. C. Willemse. Vol. 12719. Lecture Notes in Computer Science. Springer, 2021, pp. 42–59. DOI: `10.1007/978-3-030-78089-0_3` (cit. on pp. 7, 93, 206).

[18] Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. “Off-the-shelf automated analysis of liveness properties for just paths”. In: *Acta Informatica* 57.3-5 (2020), pp. 551–590. DOI: `10.1007/s00236-020-00371-w` (cit. on pp. 8, 94, 124, 131, 139, 206).

[19] Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. “Off-the-Shelf Automated Analysis of Liveness Properties for Just Paths - (Extended Abstract)”. In: *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*. Ed. by Kirstin Peters and Tim A. C. Willemse. Vol. 12719. Lecture Notes in Computer Science. Springer, 2021, pp. 182–187. DOI: `10.1007/978-3-030-78089-0_11` (cit. on p. 207).

[20] Mark Bouwman, Djurre van der Wal, Bas Luttik, Mariëlle Stoelinga, and Arend Rensink. “What is the point: Formal analysis and test generation for a railway standard”. In: *30th European Safety and Reliability Conference, ESREL 2020 and 15th Probabilistic Safety Assessment and Management Conference, PSAM15 2020*. Research Publishing Services. 2020, pp. 921–928 (cit. on pp. 71, 113, 206).

[21] Mark Bouwman, Djurre van der Wal, Bas Luttik, Mariëlle Stoelinga, and Arend Rensink. "A Case in Point: Verification and Testing of a EULYNX Interface". In: *Formal Aspects Comput.* 35.1 (2023), 2:1–2:38. DOI: 10.1145/3528207 (cit. on pp. 8, 47, 67, 71, 112, 113, 206).

[22] Mark van den Brand and Jan Friso Groote. "Software engineering: Redundancy is key". In: *Sci. Comput. Program.* 97 (2015), pp. 75–81. DOI: 10.1016/j.scico.2013.11.020 (cit. on pp. 112, 180).

[23] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. "Live Sequence Charts: An Introduction to Lines, Arrows, and Strange Boxes in the Context of Formal Verification". In: *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report.* Ed. by Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper. Vol. 3147. Lecture Notes in Computer Science. Springer, 2004, pp. 374–399. DOI: 10.1007/978-3-540-27863-4_21 (cit. on p. 182).

[24] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. "The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019.* Vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 21–39. DOI: 10.1007/978-3-030-17465-1_2 (cit. on pp. 2, 5, 11, 47, 142).

[25] Bureau of Transportation Statistics. *Transportation Fatalities by Mode.* [Online; accessed April 3, 2023]. 2023 (cit. on p. 1).

[26] Stanley Burris and Hanamantagouda P. Sankappanavar. *A course in universal algebra.* Vol. 78. Graduate texts in mathematics. Springer, 1981. ISBN: 978-0-387-90578-5 (cit. on p. 13).

[27] Ming Chai, Haifeng Wang, Tao Tang, and Hongjie Liu. "Runtime verification of train control systems with parameterized modal live sequence charts". In: *J. Syst. Softw.* 177 (2021), p. 110962. DOI: 10.1016/j.jss.2021.110962 (cit. on p. 182).

[28] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981.* Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71. DOI: 10.1007/BFb0025774 (cit. on p. 18).

[29] Sjoerd Cranen, Jan Friso Groote, and Michel A. Reniers. "A linear translation from CTL* to the first-order modal μ-calculus". In: *Theor. Comput. Sci.* 412.28 (2011), pp. 3129–3139. DOI: 10.1016/j.tcs.2011.02.034 (cit. on p. 19).

[30] Sjoerd Cranen, Bas Luttik, and Tim A. C. Willemse. “Evidence for Fixpoint Logic”. In: *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany.* Ed. by Stephan Kreutzer. Vol. 41. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 78–93. DOI: `10.4230/LIPIcs.CSL.2015.78` (cit. on pp. 11, 67, 75, 117).

[31] Fokko van Dijk, Wan J. Fokkink, Gea Kolk, Paul van de Ven, and Bas van Vlijmen. “EURIS, a Specification Method for Distributed Interlockings”. In: *Computer Safety, Reliability and Security, 17th International Conference, SAFECOMP’98, Heidelberg, Germany, October 5-7, 1998, Proceedings.* Ed. by Wolfgang D. Ehrenberger. Vol. 1516. Lecture Notes in Computer Science. Springer, 1998, pp. 296–305. DOI: `10.1007/3-540-49646-7\_23` (cit. on p. 3).

[32] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O’Hearn. “Scaling static analyses at Facebook”. In: *Commun. ACM* 62.8 (2019), pp. 62–70. DOI: `10.1145/3338112` (cit. on p. 181).

[33] Victor Dyseryn, Rob J. van Glabbeek, and Peter Höfner. “Analysing Mutual Exclusion using Process Algebra with Signals”. In: *Proceedings EXPRESS/-SOS 2017.* Ed. by Kirstin Peters and Simone Tini. Vol. 255. EPTCS. 2017, pp. 18–34. DOI: `10.4204/EPTCS.255.2` (cit. on pp. 115, 116, 118, 120, 125–127, 150).

[34] Rob van Ee and Marcel van Ee. *Ongevallen op Nederlands Spoor.* Uitgeverij De Alk BV, 1997. ISBN: 9789060130674 (cit. on p. 1).

[35] E. Allen Emerson and Chin-Laung Lei. “Modalities for Model Checking: Branching Time Logic Strikes Back”. In: *Sci. Comput. Program.* 8.3 (1987), pp. 275–306 (cit. on p. 142).

[36] E. Allen Emerson and A. Prasad Sistla. “Deciding Full Branching Time Logic”. In: *Inf. Control.* 61.3 (1984), pp. 175–201. DOI: `10.1016/S0019-9958(84)80047-9` (cit. on p. 18).

[37] Marie-Aude Esteve, Joost-Pieter Katoen, Viet Yen Nguyen, Bart Postma, and Yuri Yushtein. “Formal correctness, safety, dependability, and performance analysis of a satellite”. In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland.* Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, pp. 1022–1031. DOI: `10.1109/ICSE.2012.6227118` (cit. on p. 3).

[38] EULYNX Partners. *EULYNX modelling standard Eu.Doc.30 v4.0.* Norm. 2022 (cit. on p. 23).

[39] Alessandro Fantechi. “Twenty-Five Years of Formal Methods and Railways: What Next?” In: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers.* Ed. by Steve Counsell and Manuel Núñez. Vol. 8368. Lecture Notes

in Computer Science. Springer, 2013, pp. 167–183. DOI: 10.1007/978-3-319-05032-4_13 (cit. on p. 3).

[40] Alessio Ferrari and Maurice H. ter Beek. "Formal Methods in Railways: A Systematic Mapping Study". In: *ACM Comput. Surv.* 55.4 (2023), 69:1–69:37. DOI: 10.1145/3520480 (cit. on p. 3).

[41] Limor Fix. "Fifteen Years of Formal Property Verification in Intel". In: *25 Years of Model Checking - History, Achievements, Perspectives.* Ed. by Orna Grumberg and Helmut Veith. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 139–144. DOI: 10.1007/978-3-540-69850-0_8 (cit. on p. 181).

[42] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. "CADP 2011: a toolbox for the construction and analysis of distributed processes". In: *Int. J. Softw. Tools Technol. Transf.* 15.2 (2013), pp. 89–107. DOI: 10.1007/s10009-012-0244-z (cit. on pp. 49, 142).

[43] Hubert Garavel, Frédéric Lang, and Laurent Mounier. "Compositional Verification in Action". In: *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings.* Ed. by Falk Howar and Jiri Barnat. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 189–210. DOI: 10.1007/978-3-030-00244-2_13 (cit. on p. 49).

[44] Rob van Glabbeek, Peter Höfner, and Weiyou Wang. "Enabling Preserving Bisimulation Equivalence". In: *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference.* Ed. by Serge Haddad and Daniele Varacca. Vol. 203. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 33:1–33:20. DOI: 10.4230/LIPIcs.CONCUR.2021.33 (cit. on p. 156).

[45] Rob J. van Glabbeek. "The Linear Time - Branching Time Spectrum II". In: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings.* Ed. by Eike Best. Vol. 715. Lecture Notes in Computer Science. Springer, 1993, pp. 66–81. DOI: 10.1007/3-540-57208-2_6 (cit. on p. 49).

[46] Rob J. van Glabbeek. "The Linear Time - Branching Time Spectrum I". In: *Handbook of Process Algebra.* Ed. by Jan A. Bergstra, Alban Ponse, and Scott A. Smolka. North-Holland / Elsevier, 2001, pp. 3–99. DOI: 10.1016/b978-044482830-9/50019-9 (cit. on p. 156).

[47] Rob J. van Glabbeek. "Justness - A Completeness Criterion for Capturing Liveness Properties (Extended Abstract)". In: *FoSSaCS.* Vol. 11425. Lecture Notes in Computer Science. Springer, 2019, pp. 505–522 (cit. on pp. 118–122, 155).

[48] Rob J. van Glabbeek and Peter Höfner. "CCS: It's not fair! - Fair schedulers cannot be implemented in CCS-like languages even under progress and certain fairness assumptions". In: *Acta Inf.* 52.2-3 (2015), pp. 175–205. DOI: 10.1007/s00236-015-0221-6 (cit. on pp. 8, 116).

[49] Rob J. van Glabbeek and Peter Höfner. "Progress, Justness, and Fairness". In: *ACM Comput. Surv.* 52.4 (2019), 69:1–69:38 (cit. on pp. 94, 115, 116).

[50] Rob J. van Glabbeek, Bas Luttik, and Nikola Trcka. "Branching Bisimilarity with Explicit Divergence". In: *Fundam. Inform.* 93.4 (2009), pp. 371–392 (cit. on p. 156).

[51] Rob J. van Glabbeek, Bas Luttik, and Nikola Trcka. "Computation Tree Logic with Deadlock Detection". In: *Logical Methods in Computer Science* 5.4 (2009) (cit. on p. 156).

[52] Rob J. van Glabbeek and W. P. Weijland. "Branching Time and Abstraction in Bisimulation Semantics". In: *J. ACM* 43.3 (1996), pp. 555–600. DOI: 10.1145/233551.233556 (cit. on pp. 18, 48).

[53] Susanne Graf and Bernhard Steffen. "Compositional Minimization of Finite State Systems". In: *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings.* Ed. by Edmund M. Clarke and Robert P. Kurshan. Vol. 531. Lecture Notes in Computer Science. Springer, 1990, pp. 186–196. DOI: 10.1007/BFb0023732 (cit. on p. 49).

[54] Susanne Graf, Bernhard Steffen, and Gerald Lüttgen. "Compositional Minimisation of Finite State Systems Using Interface Specifications". In: *Formal Aspects Comput.* 8.5 (1996), pp. 607–616. DOI: 10.1007/BF01211911 (cit. on p. 49).

[55] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems.* MIT Press, 2014 (cit. on pp. 5, 11, 14–16, 20, 49).

[56] Jan Friso Groote, Alban Ponse, and Yaroslav S. Usenko. "Linearization in parallel pCRL". In: *J. Log. Algebraic Methods Program.* 48.1-2 (2001), pp. 39–70. DOI: 10.1016/S1567-8326(01)00005-4 (cit. on p. 15).

[57] Jan Friso Groote, Sebastiaan F. M. van Vlijmen, and Jan W. C. Koorn. "The safety guaranteeing system at station Hoorn-Kersenboogerd". In: *COMPASS'95 Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'.* IEEE. 1995, pp. 57–68 (cit. on p. 3).

[58] Jan Friso Groote and Tim A. C. Willemse. "Parameterised boolean equation systems". In: *Theor. Comput. Sci.* 343.3 (2005), pp. 332–369. DOI: 10.1016/j.tcs.2005.06.016 (cit. on p. 20).

[59] Jan Friso Groote and Tim A.C. Willemse. "Model-checking processes with data". In: *Sci. Comput. Program.* 56.3 (2005), pp. 251–273 (cit. on p. 149).

[60] Dennis Guck, Joost-Pieter Katoen, Mariëlle IA Stoelinga, Ted Luiten, and Judi Romijn. “Smart railroad maintenance engineering with stochastic model checking”. In: *2nd International Conference on Railway Technology: Research, Development and Maintenance* (2014) (cit. on p. 3).

[61] Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, Mohammad Reza Mousavi, and Jaco van de Pol. “Towards model checking executable UML specifications in mCRL2”. In: *ISSE* 6.1-2 (2010), pp. 83–90. DOI: `10.1007/s11334-009-0116-1` (cit. on p. 24).

[62] Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, Mohammad Reza Mousavi, Jaco van de Pol, and Osmar Marchi dos Santos. “Automated Verification of Executable UML Models”. In: *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers.* Vol. 6957. Lecture Notes in Computer Science. Springer, 2010, pp. 225–250. DOI: `10.1007/978-3-642-25271-6_12` (cit. on p. 24).

[63] Anne E. Haxthausen and Jan Peleska. “Formal Development and Verification of a Distributed Railway Control System”. In: *IEEE Trans. Software Eng.* 26.8 (2000), pp. 687–701. DOI: `10.1109/32.879808` (cit. on p. 3).

[64] Anne E. Haxthausen and Jan Peleska. “Model Checking and Model-Based Testing in the Railway Domain”. In: *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015.* Ed. by Rolf Drechsler and Ulrich Kühne. Springer, 2015, pp. 82–121. DOI: `10.1007/978-3-658-09994-7_4` (cit. on p. 3).

[65] Anne E. Haxthausen, Jan Peleska, and Ralf Pinger. “Applied Bounded Model Checking for Interlocking System Designs”. In: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers.* Ed. by Steve Counsell and Manuel Núñez. Vol. 8368. Lecture Notes in Computer Science. Springer, 2013, pp. 205–220. DOI: `10.1007/978-3-319-05032-4_16` (cit. on p. 3).

[66] Matthew Hennessy and Robin Milner. “Algebraic Laws for Nondeterminism and Concurrency”. In: *J. ACM* 32.1 (1985), pp. 137–161. DOI: `10.1145/2455.2460` (cit. on pp. 18, 165).

[67] Christoph Hilken, Jan Peleska, and Robert Wille. “A Unified Formulation of Behavioral Semantics for SysML Models”. In: *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, 9-11 February, 2015.* Ed. by Slimane Hammoudi, Luís Ferreira Pires, Philippe Desfray, and Joaquim Filipe. SciTePress, 2015, pp. 263–271. DOI: `10.5220/0005241602630271` (cit. on p. 24).

[68] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004. ISBN: 978-0-321-22862-8 (cit. on p. 24).

[69] Linh Vu Hong, Anne E. Haxthausen, and Jan Peleska. “Formal modelling and verification of interlocking systems featuring sequential release”. In: *Sci. Comput. Program.* 133 (2017), pp. 91–115. DOI: 10.1016/j.scico.2016.05.010 (cit. on p. 3).

[70] Yi-Ling Hwong, Jeroen J. A. Keiren, Vincent J. J. Kusters, Sander J. J. Leemans, and Tim A. C. Willemse. “Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider”. In: *Sci. Comput. Program.* 78.12 (2013), pp. 2435–2452. DOI: 10.1016/j.scico.2012.11.009 (cit. on p. 3).

[71] Phillip James, Faron Moller, Nguyen Hoang Nga, Markus Roggenbach, Steve A. Schneider, and Helen Treharne. “Techniques for modelling and verifying railway interlockings”. In: *Int. J. Softw. Tools Technol. Transf.* 16.6 (2014), pp. 685–711. DOI: 10.1007/s10009-014-0304-7 (cit. on p. 3).

[72] Phillip James and Markus Roggenbach. “Automatically Verifying Railway Interlockings using SAT-based Model Checking”. In: *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 35 (2010). DOI: 10.14279/tuj.eceasst.35.547 (cit. on p. 3).

[73] Jeroen J. A. Keiren and Martijn Klabbers. “Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2”. In: *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 53 (2012). DOI: 10.14279/tuj.eceasst.53.793 (cit. on p. 47).

[74] Soon-Kyeong Kim and David A. Carrington. “A Formal Model of the UML Metamodel: The UML State Machine and Its Integrity Constraints”. In: *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings.* Vol. 2272. Lecture Notes in Computer Science. Springer, 2002, pp. 497–516. DOI: 10.1007/3-540-45648-1_26 (cit. on p. 24).

[75] Dexter Kozen. “Results on the propositional μ-calculus”. In: *Theoretical Computer Science* 27.3 (1982), pp. 333–354. ISSN: 16113349. DOI: 10.1007/BFb0012782 (cit. on p. 19).

[76] Sabine Kuske. “A Formal Semantics of UML State Machines Based on Structured Graph Transformation”. In: *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings.* Vol. 2185. Lecture Notes in Computer Science. Springer, 2001, pp. 241–256. DOI: 10.1007/3-540-45441-1_19 (cit. on p. 24).

[77] Leslie Lamport. “A New Solution of Dijkstra’s Concurrent Programming Problem”. In: *Commun. ACM* 17.8 (1974), pp. 453–455. DOI: 10.1145/361082.361093 (cit. on p. 155).

[78] Frédéric Lang. “Refined Interfaces for Compositional Verification”. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006.* Ed. by Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge. Vol. 4229. Lecture Notes in Computer Science. Springer, 2006, pp. 159–174. DOI: `10.1007/11888116_13` (cit. on p. 49).

[79] Frédéric Lang, Radu Mateescu, and Franco Mazzanti. “Compositional verification of concurrent systems by combining bisimulations”. In: *Formal Methods Syst. Des.* 58.1-2 (2021), pp. 83–125. DOI: `10.1007/s10703-021-00360-w` (cit. on p. 49).

[80] Diego Latella, István Majzik, and Mieke Massink. “Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker”. In: *Formal Asp. Comput.* 11.6 (1999), pp. 637–664. DOI: `10.1007/s001659970003` (cit. on p. 24).

[81] Maurice Laveaux, Wieger Wesselink, and Tim A. C. Willemse. “On-The-Fly Solving for Symbolic Parity Games”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II.* Ed. by Dana Fisman and Grigore Rosu. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 137–155. DOI: `10.1007/978-3-030-99527-0_8` (cit. on p. 67).

[82] Maurice Laveaux and Tim A. C. Willemse. “Decompositional Minimisation of Monolithic Processes”. In: *CoRR* abs/2012.06468 (2020). arXiv: `2012.06468` (cit. on p. 56).

[83] Maurice Laveaux and Tim A. C. Willemse. “Decomposing Monolithic Processes in a Process Algebra with Multi-actions”. In: *Proceedings 14th Interaction and Concurrency Experience, ICE 2021, Online, 18th June 2021.* Ed. by Julien Lange, Anastasia Mavridou, Larisa Safina, and Alceste Scalas. Vol. 347. EPTCS. 2021, pp. 57–76. DOI: `10.4204/EPTCS.347.4` (cit. on pp. 16, 17, 49, 54–56, 68).

[84] Johan Lilius and Ivan Paltor. “vUML: A Tool for Verifying UML Models”. In: *The 14th IEEE International Conference on Automated Software Engineering, ASE 1999, Cocoa Beach, Florida, USA, 12-15 October 1999.* IEEE Computer Society, 1999, pp. 255–258. DOI: `10.1109/ASE.1999.802301` (cit. on p. 24).

[85] Johan Lilius and Ivn Porres Paltor. “The semantics of UML state machines”. In: (1999) (cit. on p. 24).

[86] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong. “A Formal Semantics for Complete UML State Machines with Communications”. In: *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013.*

Proceedings. Vol. 7940. Lecture Notes in Computer Science. Springer, 2013, pp. 331–346. DOI: 10.1007/978-3-642-38613-8_23 (cit. on p. 24).

[87] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of abstract data types.* Wiley, 1996. ISBN: 978-0-471-95067-7 (cit. on p. 13).

[88] Bjørnar Luteberget and Christian Johansen. “Efficient verification of railway infrastructure designs against standard regulations”. In: *Formal Methods Syst. Des.* 52.1 (2018), pp. 1–32. DOI: 10.1007/s10703-017-0281-z (cit. on p. 3).

[89] Robin Milner. *Communication and concurrency.* PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2 (cit. on p. 160).

[90] Mohammad Reza Mousavi, Michel A. Reniers, and Jan Friso Groote. “Notions of bisimulation and congruence formats for SOS with data”. In: *Inf. Comput.* 200.1 (2005), pp. 107–147. DOI: 10.1016/j.ic.2005.03.002 (cit. on pp. 162–164).

[91] Thomas Neele, Tim A. C. Willemse, and Jan Friso Groote. “Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting”. In: *FACS*. Vol. 11222. Lecture Notes in Computer Science. Springer, 2018, pp. 216–236 (cit. on p. 48).

[92] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. “How Amazon web services uses formal methods”. In: *Commun. ACM* 58.4 (2015), pp. 66–73. DOI: 10.1145/2699417 (cit. on p. 181).

[93] Rocco De Nicola and Rosario Pugliese. “A Process Algebra Based on LINDA”. In: *COORDINATION*. Vol. 1061. Lecture Notes in Computer Science. Springer, 1996, pp. 160–178. DOI: 10.1007/3-540-61052-9_45 (cit. on p. 175).

[94] Mogens Nielsen, Klaus Havelund, Kim Ritter Wagner, and Chris George. “The RAISE Language, Method and Tools”. In: *Formal Aspects Comput.* 1.1 (1989), pp. 85–114. DOI: 10.1007/BF01887199 (cit. on p. 3).

[95] Object Managament Group. *OMG Unified Modeling Language, version 2.5.1.* Norm. 2017 (cit. on pp. 4, 27, 30).

[96] Object Managament Group. *Systems Modeling Language (SysML) v2 Request For Proposal (RFP). OMG Document: ad/2017-12-02.* Norm. 2017 (cit. on p. 182).

[97] Object Managament Group. *OMG Systems Modeling Language, version 1.6.* Norm. 2019 (cit. on p. 4).

[98] Object Managament Group. *Precise Semantics of UML State Machines (PSSM), version 1.0.* Norm. 2019 (cit. on p. 25).

[99] Ivan Paltor and Johan Lilius. "Formalising UML State Machines for Model Checking". In: *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*. 1999, pp. 430–445. DOI: 10.1007/3-540-46852-8_31 (cit. on p. 24).

[100] David M. R. Park. "Concurrency and Automata on Infinite Sequences". In: *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*. Ed. by Peter Deussen. Vol. 104. Lecture Notes in Computer Science. Springer, 1981, pp. 167–183. DOI: 10.1007/BFb0017309 (cit. on pp. 18, 48, 156).

[101] Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck. "AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties". In: *11th Annual International Conference on New Technologies of Distributed Systems, NOTERE 2011, Paris, France, 9-13 May 2011*. IEEE, 2011, pp. 1–10. DOI: 10.1109/NOTERE.2011.5957992 (cit. on p. 24).

[102] Jan Peleska and Wen-ling Huang. "Industrial-Strength Model-Based Testing of Safety-Critical Systems". In: *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. Ed. by John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou. Vol. 9995. Lecture Notes in Computer Science. 2016, pp. 3–22. DOI: 10.1007/978-3-319-48989-6_1 (cit. on p. 24).

[103] Jan Peleska, Wen-ling Huang, and Uwe Schulze. *Test automation support. Technical report D34.1, COMPASS Comprehensive Modelling for Advanced Systems of Systems*. Ed. by John S. Fitzgerald and Peter Gorm Larsen. http://www.compass-research.eu/deliverables.html. 2013 (cit. on p. 24).

[104] Gary L. Peterson. "Myths About the Mutual Exclusion Problem". In: *Inf. Process. Lett.* 12.3 (1981), pp. 115–116. DOI: 10.1016/0020-0190(81)90106-X (cit. on pp. 116, 155).

[105] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cit. on p. 18).

[106] PTC. *ASAL structured action language (SySim*. [Online; accessed April 7, 2023]. 2023 (cit. on p. 30).

[107] Daniela Remenska, Jeff Templon, Tim A. C. Willemse, Philip Homburg, Kees Verstoep, Adrian Casajus Ramo, and Henri E. Bal. "From UML to Process Algebra and Back: An Automated Approach to Model-Checking Software Design Artifacts of Concurrent Systems". In: *NASA Formal Methods*. Vol. 7871. Lecture Notes in Computer Science. Springer, 2013, pp. 244–260 (cit. on p. 25).

[108] Daniela Remenska, Tim A. C. Willemse, Kees Verstoep, Jeff Templon, and Henri E. Bal. “Using model checking to analyze the system behavior of the LHC production grid”. In: *Future Gener. Comput. Syst.* 29.8 (2013), pp. 2239–2251. DOI: `10.1016/j.future.2013.06.004` (cit. on p. 47).

[109] A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010. DOI: `10.1007/978-1-84882-258-0` (cit. on p. 175).

[110] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. “Semantic Foundations of Concurrent Constraint Programming”. In: *POPL.* ACM Press, 1991, pp. 333–352. DOI: `10.1145/99583.99627` (cit. on p. 175).

[111] Timm Schäfer, Alexander Knapp, and Stephan Merz. “Model checking UML state machines and collaborations”. In: *Electron. Notes Theor. Comput. Sci.* 55.3 (2001), pp. 357–369. DOI: `10.1016/S1571-0661(04)00262-2` (cit. on p. 24).

[112] Steve A. Schneider and Helen Treharne. “CSP theorems for communicating B machines”. In: *Formal Aspects Comput.* 17.4 (2005), pp. 390–422. DOI: `10.1007/s00165-005-0076-7` (cit. on p. 3).

[113] Danielle Stewart, Jing Liu, Darren D. Cofer, Mats Per Erik Heimdahl, Michael W. Whalen, and Michael Peterson. “AADL-Based safety analysis using formal methods applied to aircraft digital systems”. In: *Reliab. Eng. Syst. Saf.* 213 (2021), p. 107649. DOI: `10.1016/j.ress.2021.107649` (cit. on p. 3).

[114] VDE. *Electric signalling systems for railways – Part200: Safe transmission protocol according to DIN EN50159 (DIN VDE V 0831-159).* DIN VDE V 0831-200. Norm. June 2015 (cit. on p. 72).

[115] Hongli Wang, Deming Zhong, Tingdi Zhao, and Fuchun Ren. “Integrating Model Checking With SysML in Complex System Safety Analysis”. In: *IEEE Access* 7 (2019), pp. 16561–16571. DOI: `10.1109/ACCESS.2019.2892745` (cit. on p. 24).

[116] Wieger Wesselink and Tim A. C. Willemse. “Evidence Extraction from Parameterised Boolean Equation Systems”. In: *Proceedings of the 3rd International Workshop on Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2018) affiliated with the International Joint Conference on Automated Reasoning (IJCAR 2018), Oxford, UK, July 18, 2018.* Ed. by Christoph Benzmüller and Jens Otten. Vol. 2095. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 86–100 (cit. on pp. 11, 67, 75, 117).

[117] Wikimedia Commons. *File:Facing points Broomhill.jpg.* [Online; accessed July 9, 2021]. 2021 (cit. on p. 94).

Summary

This thesis, titled “Supporting Railway Standardisation with Formal Verification”, has been written in the context of the FormaSig project. This project involves a collaboration between academia and railway infrastructure managers to use formal methods to support the development of the EULYNX standard. This standard uses the SysML modelling language to define the interfaces of the various components of a signalling system (signal, point, level crossing, etcetera). The aim of the FormaSig project is to define a formal interpretation of the standard such that delivered components conforming to the standard provably satisfy a collection of safety properties. The idea is to associate with each SysML model a formal mCRL2 model. Then mCRL2’s model checker can be used to establish that the model satisfies the required safety properties, and automated model-based test technology can be used to thoroughly test compliance to the model of actual implementations. This thesis focusses on the aspects of formalisation and verification.

The first contribution is a formalisation of SysML in mCRL2. The approach taken is to generically define the structure and semantics of SysML state machines in mCRL2. This generic model forms the basis of a translation tool that encodes a set of concrete SysML diagrams in the mCRL2 data language.

A challenge in model checking is scalability; the state space of a model tends to scale exponentially with the number of parallel components. Our second contribution is an inventory of techniques to reduce the size of the state space induced by mCRL2 models obtained through our SysML to mCRL2 translation. We found that the most effective technique for our models is compositional minimisation. With this technique the state space of the entire (monolithic) model is not computed in one go. Instead, the model is first split into components; in our case a SysML state machine is a component. The state space of each component is computed and then minimised modulo an equivalence relation. These minimised state spaces are finally combined to construct the state space of the entire model. The existing theory is extended to support branching bisimulation minimisation, greatly increasing its effectiveness.

The next contribution of this thesis is to discuss the application of the aforementioned translation and scalability techniques to several concrete EULYNX interfaces. For each interface (safety) requirements are formulated and verified using the mCRL2 toolset. The verification efforts identified errors and omissions in the standard and this led to improvements of the standard. The case studies

have also led to improvements of the toolchain.

The case studies reveal that we sometimes want to verify liveness requirements stating that something good always eventually happens. For such requirements, assumptions are often needed on how events are scheduled. These range from very basic assumptions to rather strong assumptions that may not be realistic. A recently introduced notion is that of 'justness'. In short, it is the assumption that once an action is enabled that stems from a set of parallel components then one (or more) of these components will eventually partake in an action. This seems to be a realistic assumption that is just strong enough for many liveness requirements. This thesis contributes a way to verify liveness requirements with a justness assumption for mCRL2 models. Moreover, it is applied in the aforementioned case studies.

Besides justness, the cases studies reveal another need: the concept of global variables. The semantics of mCRL2 is action-based and thus abstracts from the contents of a state; states are only distinguished by their transitions. The logic (the modal μ-calculus in the case of the mCRL2 toolset) also only refers to transitions. For models that are derived from state based formalisms (such as SysML) it can be desirable to also consider the contents of a state. For some requirements for FormaSig models we need to specify that the system always (or never) ends up in a specific SysML state. The final contribution in this thesis is a process algebra with global variables and a logic allowing references to these variables. This enables the inspection of these global variables during verification and thus allows for requirements referencing both the transitions and the contents of states. Moreover, in some settings, it allows for a more natural form of communication between parallel components (compared to message passing). The theory is applied to formulate requirements in FormaSig.

Samenvatting

De titel van dit proefschrift kan worden vertaald als “Het Gebruik van Formele Methoden in de Ondersteuning van Standaardisatie Binnen de Spoorwegen”. Dit proefschrift is tot stand gekomen binnen de context van het FormaSig project. Dit project behelst een samenwerking van de academische wereld en de spoorwegen om met behulp van formele methoden de ontwikkeling van de EULYNX standaard te ondersteunen. Deze standaard gebruikt de SysML modelleertaal om de interfaces tussen verscheidene onderdelen in het seinwezen te specificeren (seinen, wissels, spoorwegovergangen, et cetera). Het doel van het FormaSig project is een formele interpretatie van de standaard te definiëren zodat geleverde componenten die aan de standaard voldoen, aantoonbaar voldoen aan een verzameling veiligheidseigenschappen. Het idee is om van elk SysML model een formeel mCRL2 model af te leiden. Vervolgens kunnen de verificatietools van mCRL2 worden gebruikt om te bewijzen dat het model voldoet aan de veiligheidseisen en kunnen test cases afgeleid uit het model worden gebruikt om de conformiteit van daadwerkelijke implementaties met het model grondig te testen. Dit proefschrift focust op de formalisatie en verificatie aspecten.

De eerste bijdrage is een formalisatie van SysML in mCRL2. In de gekozen aanpak definiëren we de structuur en semantiek van SysML state machines op generieke wijze in mCRL2. Dit generieke model vormt de basis voor een vertaaltool die een set concrete SysML diagrammen in de datataal van mCRL2 kan encoderen.

Een uitdaging bij formele verificatie is schaalbaarheid; de toestandsruimte van een model schaalt vaak exponentieel met het aantal parallelle componenten. De tweede bijdrage is een inventarisatie van technieken om de omvang van de toestandsruimte van mCRL2-modellen, verkregen door onze vertaling van SysML naar mCRL2, te verkleinen. Het blijkt dat de meest effectieve techniek voor onze modellen compositionele minimalisatie is. Bij deze techniek wordt de toestandsruimte van het gehele (monolithische) model niet in één keer berekend. In plaats daarvan wordt het model eerst opgesplitst in componenten; in ons geval is een SysML state machine een component. De toestandsruimte van elke component wordt berekend en vervolgens geminimaliseerd volgens een equivalentierelatie. Deze geminimaliseerde toestandsruimten worden uiteindelijk gecombineerd om de toestandsruimte van het gehele model te construeren. De bestaande theorie is uitgebreid om branching bisimulatie minimalisatie te ondersteunen, waardoor de effectiviteit ervan sterk toeneemt.

De volgende bijdrage van dit proefschrift is de toepassing van de bovengenoemde vertaal- en schaalbaarheidstechnieken op verschillende concrete EULYNX interfaces en het bespreken van de lessen die daaruit volgen. Voor elke interface zijn (veiligheids)eisen geformuleerd en geverifieerd met behulp van de mCRL2 toolset. Verificatie heeft fouten en omissies in de standaard aan het licht gebracht en dit heeft geleid tot verbeteringen van de standaard. De casestudy's hebben ook geleid tot verbeteringen van de toolchain.

Uit de casestudy's blijkt dat we soms liveness eisen willen verifiëren die stellen dat er uiteindelijk altijd iets wenselijks gebeurt. Voor dergelijke eisen zijn vaak aannames nodig over de volgorde van gebeurtenissen. Deze variëren van zeer rudimentaire aannames tot vrij sterke aannames die wellicht niet realistisch zijn. Een recent geïntroduceerd begrip is dat van *justness*. Kort gezegd is dit de aanname dat, zodra een actie mogelijk is die voortkomt uit een verzameling parallelle componenten, één (of meer) van deze componenten uiteindelijk zal deelnemen aan een actie. Dit lijkt een realistische aanname te zijn die net sterk genoeg is voor veel liveness eisen. De bijdrage in deze dissertatie is een methode die verificatie van liveness eisen met een *justness* aanname mogelijk maakt voor mCRL2 modellen. Bovendien wordt het toegepast in de bovengenoemde casestudy's.

Naast *justness* brengen de casestudy's nog een andere behoefte aan het licht: het concept van globale variabelen. De semantiek van mCRL2 is gebaseerd op acties en abstraheert dus van de inhoud van een toestand; toestanden worden alleen onderscheiden door hun transities. De logica (de modale μ-calculus in het geval van de mCRL2 toolset) verwijst ook alleen naar transities. Voor modellen die zijn afgeleid van toestandsgebaseerde formalismen (zoals SysML) kan het wenselijk zijn ook de inhoud van een toestand in beschouwing te nemen. Bij sommige eisen voor EULYNX modellen moeten we uitdrukken dat het systeem altijd (of nooit) in een specifieke SysML-toestand terechtkomt. De laatste bijdrage in dit proefschrift is een procesalgebra met globale variabelen en een logica die verwijzingen naar deze variabelen toestaat. Dit maakt de inspectie van deze globale variabelen tijdens verificatie mogelijk en maakt het dus mogelijk eisen te formuleren die verwijzen naar zowel de transities als de inhoud van toestanden. Bovendien maakt het in sommige settings een meer natuurlijke vorm van communicatie mogelijk tussen parallelle componenten (in vergelijking met message passing). De theorie wordt toegepast om eisen te formuleren in FormaSig.

Curriculum Vitae

Mark Bouwman was born in Dordrecht, where he grew up. After high school he moved to Leuven to study theology. The following year he switched to the Computer Science bachelor program at Eindhoven University of Technology. In his last year he was also praeses/chairman of the Navigators student association. After finishing his BSc degree he continued at the same university with the Computer Science master program. During his studies he showed particular interest in information security technology, program verification and model checking. The research for his master's thesis was carried out at Siemens under the supervision of dr. Bob Janssen (Siemens) and dr.ir. Bas Luttik (TU/e). The research focussed on both model checking and model-based testing of railway interlocking systems.

At the invitation of Bas Luttik, Mark started as a PhD candidate at the Formal System Analysis group in Eindhoven. His project was funded by ProRail and DB Netz AG and focussed on developing verification technology for the European EULYNX project. During his time as a PhD candidate Mark made several side steps to research on global variables in process algebras, justness and efficient term rewriting. His dissertation is titled "Supporting Railway Standardisation with Formal Verification".

List of Publications

Mark Bouwman has the following publications.

Journal Publications

- Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. “Off-the-shelf automated analysis of liveness properties for just paths”. In: *Acta Informatica* 57.3-5 (2020), pp. 551–590. DOI: `10.1007/s00236-020-00371-w`

- Mark Bouwman, Djurre van der Wal, Bas Luttik, Mariëlle Stoelinga, and Arend Rensink. “A Case in Point: Verification and Testing of a EULYNX Interface”. In: *Formal Aspects Comput.* 35.1 (2023), 2:1–2:38. DOI: `10.1145/3528207`

Conference Proceedings

- Mark Bouwman, Bob Janssen, and Bas Luttik. “Formal Modelling and Verification of an Interlocking Using mCRL2”. In: *Formal Methods for Industrial Critical Systems - 24th International Conference, FMICS 2019, Amsterdam, The Netherlands, August 30-31, 2019, Proceedings.* Ed. by Kim Guldstrand Larsen and Tim A. C. Willemse. Vol. 11687. Lecture Notes in Computer Science. Springer, 2019, pp. 22–39. DOI: `10.1007/978-3-030-27008-7_2`

- Mark Bouwman, Bas Luttik, Wouter Schols, and Tim A. C. Willemse. “A process algebra with global variables”. In: *Proceedings Combined 27th International Workshop on Expressiveness in Concurrency and 17th Workshop on Structural Operational Semantics, EXPRESS/SOS 2020, and 17th Workshop on Structural Operational Semantics, 31 August 2020.* Ed. by Ornela Dardha and Jurriaan Rot. Vol. 322. EPTCS. 2020, pp. 33–50. DOI: `10.4204/EPTCS.322.5`

- Mark Bouwman, Djurre van der Wal, Bas Luttik, Mariëlle Stoelinga, and Arend Rensink. “What is the point: Formal analysis and test generation for a railway standard”. In: *30th European Safety and Reliability Conference, ESREL 2020 and 15th Probabilistic Safety Assessment and Management Conference, PSAM15 2020.* Research Publishing Services. 2020, pp. 921–928

- Mark Bouwman, Bas Luttik, and Djurre van der Wal. “A Formalisation of SysML State Machines in mCRL2”. In: *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings.* Ed. by Kirstin Peters and Tim A. C. Willemse. Vol. 12719. Lecture Notes in Computer Science. Springer, 2021, pp. 42–59. DOI: `10.1007/978-3-030-78089-0_3`

- Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. “Off-the-Shelf Automated Analysis of Liveness Properties for Just Paths - (Extended Abstract)”. In: *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings.* Ed. by Kirstin Peters and Tim A. C. Willemse. Vol. 12719. Lecture Notes in Computer Science. Springer, 2021, pp. 182–187. DOI: `10.1007/978-3-030-78089-0_11`

- Mark Bouwman, Maurice Laveaux, Bas Luttik, and Tim A. C. Willemse. “Decompositional Branching Bisimulation Minimisation of Monolithic Processes”. In: *Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings.* Ed. by Silvia Lizeth Tapia Tarifa and José Proença. Vol. 13712. Lecture Notes in Computer Science. Springer, 2022, pp. 161–182. DOI: `10.1007/978-3-031-20872-0\_10`

Master Thesis

- Mark Bouwman. “A model-based test platform for rail signalling systems”. Master’s thesis. Eindhoven University of Technology, 2019

Technical Reports (Non-Refereed)

- Mark Bouwman and Rick Erkens. “Term Rewriting Based On Set Automaton Matching”. In: *CoRR* abs/2202.08687 (2022). arXiv: `2202.08687`

Titles in the IPA Dissertation Series since 2020