Department of Mathematics and Computer Science
Formal System Analysis Research Group

An approachable language for formal requirements

Master's Thesis

Kevin Antonius Henricus Maria Nogarede
0907625

Master:	Embedded Systems
Department:	Mathematics and Computer Science
Research Group:	Formal System Analysis
In collaboration with:	ASML
Committee:	
(TU/e) Supervisor:	dr.ir. T.A.C. Willemse
(ASML) Supervisor:	dr.ir. R.R.H. Schiffelers
	Prof.dr.ir. J.F. Groote
Report ID:	version 1.2.3

Eindhoven, December 2019

Abstract

Formal system verification is a mathematical technique for establishing whether a process meets certain design requirements. Typically, such techniques require notation in academic languages which are difficult for engineers to write and interpret. We aim to develop a new DSL for formalizing requirements that dramatically lowers the barrier of entry by introducing notation and concepts that are intuitively understandable yet still amenable to automated verification. We prove the correctness of a translation from this new DSL to mCRL2, a well-established tool in formal model analysis. The applicability of the new language is assessed via user experience studies.

Executive summary

Formal system verification is a mathematical technique for establishing whether a (software) system meets certain design requirements. Typically, such requirements are phrased in academic languages such as temporal logics (e.g. CTL or LTL), or modal languages (e.g. the modal μ-calculus). A downside of such languages is that these are hard to understand and not tailored to the problem domain, thus preventing their use in practice. We aim to develop a new way of formalizing requirements that is (1) more accessible to engineers, (2) sufficiently expressive, and (3) amenable to automated verification. The applicability of the new way of formalizing requirements is assessed via user experience studies.

By establishing a broad knowledge-base of the kinds of requirements that are being written by students, as well as what templates they use to convey these requirements in natural language, what verbiage they use, and what paradigms are more popular than others, we find that students do not tend to deviate far from the template requirements they were originally taught, suggesting that they are not comfortable enough to write more complex requirements, or to leverage techniques other than those they were explicitly taught.

When analyzing what kind of mistakes they make in translating requirements to formal specifications, it is notable that even though the students do stick to the template requirements they were taught, they do still make simple mistakes which the course material instructed them not to make. This suggests that the μ-calculus language is not particularly conducive to human interpretation, and as such, students are not able to find the mistakes they make even when they have a decent understanding of the theory.

Based on these findings, we conclude that the modal μ-calculus that mCRL2 uses is not accessible enough for use in practice. Additionally, there is a need for a more friendly alternative which is easier to use than the μ-calculus and more broadly applicable than the alternatives found in the PSP or Remenska frameworks.

As such, we have created a language that hides some of the complexity of the modal μ-calculus, by introducing more meaningful (that is, humanly interpretable) operators that are better tailored to constructing the types of requirements these students often write. This new language removes the complex interplay of operators and contexts as seen in the modal μ-calculus, and replaces it with simple syntactical rules that guide the user into writing the correct requirement.

This is backed up by the user experience studies. Given around 15 minutes of explanation of the new

language, none of the testers wrote syntactically invalid or semantically meaningless requirements; something which the students could not achieve using modal μ-calculus with around 7 weeks of tuition and project supervision. Additionally, while the speed and adaptability of the testers varied greatly, the requirements they managed to write in the time available to them reflected the intention of the given requirements. This in contrast to the students who could only manage 58%. While the user experience studies were only done with 4 people, the gap between the results is significant enough to be convincing.

Finally, we provide a translation from our new language to the modal μ-calculus that mCRL2 uses, and prove that the semantics of our langauge are equivalent to the semantics of any translated formula in the modal μ-calculus. This is particularly useful because mCRL2 is a mature and well-established toolset for formal model analysis.

Preface

This thesis, named 'An approachable language for formal requirements', concludes the research that I have performed as part of my graduation project. This project serves to finalize my Master Embedded Systems study, to obtain the Master of Science title. I have been working towards this thesis since September 2018, going fulltime on Februari 2019, until now, December 2019.

The project was done in collaboration with ASML. The sentiment of a few ASML engineers towards requirement formalization languages has sparked the idea for this investigation, and together with Jan Friso and Tim we were able to formulate a research question. The sheer amount of scope, flexibility, and creative license that was given to me has made this by far the most challenging and interesting project to date.

I would like to thank my supervisors of this graduation project, Jan Friso Groote, Tim Willemse, and Ramon Schiffelers. Jan Friso for introducing this project to me, and being an invaluable source of information and opinions throughout the project, as well as introducing me to Tim and Ramon. Your guidance, as well as your insights, remarks, and attitude towards the current state of formal verification have shaped my perception on this subject, and have made me appreciate this project as far more than just an intellectual challenge. Inviting me to the Formal System Analysis group's colloquia allowed me to get a good grasp on the goals and achievements of the research group, and has on multiple occasions inspired me on ways to give shape to this project. Tim for being a great target to bounce ideas off of, and guiding me towards a finished project. The amount of support you have provided to me, in reaching a formal proof as well as help on the theory behind the formalizations, translations, and the proof reading, is very appreciated. Finally, Ramon, thank you for the opportunity to work on this project with ASML, and with all the great people I met here. You have given me the freedom to work as suited me best, and helped me take responsiblity for my own project.

Thank you to all the people I met at ASML, students and employees both, for making the experience of working alongside others a truly memorable experience. Bharat, Sander, Hanze, Sam, Susnehalatha, Nan, and Ferry, thank you for our brainstorm sessions, the thursday nights, and the enjoyable atmosphere. Sven, I really appreciate the time and effort you have put into our discussions. Our talks were instrumental in getting an overview of the problems I was trying to solve, and helped me recognize different perspectives. Without you, the language would be significantly less usable. Maikel, thank you for taking the time to help me figure out how to get usable results from user testing, as well as your willingness to believe in the value and success of this project.

I also really appreciate the time that Nontas, Dennis, Olav, and Mark have invested in testing the new language, having to deal with what is by any standard a very short timeframe to familiarize themselves with a completely new language and still produce useable results for me despite this burden.

Finally, I am eternally grateful for the unconditional love and support from my family.

Kevin
Eindhoven, December 2019

Contents

Chapter 1

Introduction

With computing power becoming less expensive every year [1], the industry is increasingly able to produce more complex and massive software systems. While these more complex systems allow us to leverage the power of computers to new levels, the complexity in itself is a very big cost that brings with it new challenges.

Exacerbating the issue is the rise of techniques such as parallel computing and data-driven design philosophies. With the declining growth of processor speed and transistor densities as well as the increasingly low cost of computing power in itself, the primary direction of the search for higher performance nowadays is towards exploiting various levels of parallelism [2]. These evolutions in the creation of software have transformed relatively simple sequential software systems into beasts of complexity and multitudes of interacting components.

Finding bugs in highly parallel programs has traditionally been exceptionally difficult due to the non-deterministic nature of such systems; differences in order of execution by means of, e.g., thread interleaving, signal delivery timings, and I/O events can cause an exponential number of possible execution traces. Artifacts such as race conditions and data races may mean that only some of these exhibit incorrect behavior.

Even worse is that pinpointing bugs 'in the wild' is often incredibly more complex because the parallelism allows all but the buggy component to continue operating as if nothing happened, such that the effects of the bug may sometimes manifest significantly later in the execution than the cause of the bug.

Determining whether a given (parallel or otherwise) non-trivial system is correct (that is; behaves according to a set of requirements) with conventional methods such as unit testing [3] and functional testing [4] are patently impractical; ensuring that all relevant behavior under all possible conditions is tested is infeasible.

That is not to say that these testing techniques do not have their uses. They are powerful in their own right in identifying programming issues and can increase reliability when done appropriately as shown when evaluating frameworks like Test Driven Development [5].

A solution presents itself in formal methods. These provide a framework for rigorous and exhaustive verification [6]. While formal methods are not a holy grail—no technique will save you from writing a bad or incomplete specification, and formal methods have intrinsic shortcomings of their own—there is indirect, qualitative evidence that using formal methods is beneficial in every step of production; specifications are more thorough, concise, and convincing, development is less risky, production costs may decrease, both coding and maintenance are easier, and the resulting system contains fewer architectural errors [7]. However, quantitative studies into the effects of using formal methods are difficult to find.

Formal methods come in a broad variety of approaches.

model checking whereby for a given model of a system, it is automatically and exhaustively checked whether the model satisfies a given specification. Notable examples include BLAST, CADP, mCRL2, NuSMV, and Prism, which verify properties described in temporal logics, such as LTL, PSL, SVA, CTL, and modal μ-calculus.

deductive verification where properties are verified using axiomatic deductive reasoning. These generally come in the form of theorem provers such as Prover9/Mace4, HOL, ACL2, Isabelle, Coq, and PVS, and satisfiability modulo theory (SMT) solvers such as Yices and Z3.

program synthesis where the program is automatically generated (deductively, constructively, or supervisory) from the specification itself, usually given in a logical calculus. Notable examples are Supervisory Control Synthesis.

static program analysis also known as extended static checking, where the code is analysed directly using a broad range of techniques.

formal equivalence checking also known as electronic design automation, where a reference system is given as input and a functionally equivalent, optimised (along some metric) system is returned.

However, formal methods can be rather difficult to work with. These methods are primarily designed for academic study rather than any business application, and as such tend to suffer from obscure and intricate syntaxes. Additionally, their formality is used to solve the problem of ambiguity in the natural languages, but this requires the user to reason in a radically different manner than they might be used to. As an example, we may require of an electric car that "when the throttle is released, no more power must be sent to the motors". This would be a perfectly reasonable statement to tell fellow engineers, but formally this may be translated in ways that intuitively make no sense: should the motors *never again* be sent any power after the throttle is released once?

Attempts have been made to allow untrained people to understand these intricacies. A particularly noteworthy attempt is made by Dwyer et al. in constructing the PSP framework [8]. They describe a framework in which experts have created templates of formal structures with detailed descriptions of the subtleties of that structure. A relatively lay person may then use these descriptions to pick suited templates and fill these in without having to gain the expert knowledge necessary to figure out the complexities themselves.

Remenska [9][10] builds on this work by extending the knowledge base introduced by Dwyer et al. and providing an interactive experience that helps people determine exactly which template they need by asking for the intent and needs of the user.

However, these attempts have shortcomings of their own: a tradeoff must be made between the number of templates and the usability of the framework. Writing a template for every requirement ever devised is impractical, and so is trying to find the correct template when there are so many to pick from. A better solution would be to remove (or at least reduce) the need for expert knowledge in the first place. As such, we attempt to create a new language that is more accessible than popular languages such as the modal μ-calculus.

1.1 Research Objectives

First, we must prove the motivation for the project; the assertion that current formal verification techniques are inadequately accessible:

Research question 0 Can we determine that there is a need for a more approachable language to facilitate specifying and verifying formal requirements, or does there already exist a sufficient alternative?

Having adequately determined that there is indeed a need for a simpler, more intuitive language to write specifications with, we should then investigate what exactly it is that would make a specification language usable to engineers;

Research question 1 What issues do engineers currently face when trying to translate requirements into formal specifications?

Once we have determined exactly what issues are present for engineers, we have to try to figure out why exactly it is that these issues are present; in order to solve the problem, we should first know the cause. When we know what the causes are, it is then imperative to try to either resolve the causes, or minimize the impact that the causes may have.

Research question 2 What are the causes of the issues that present themselves in formalizing requirements?

Research question 3 How, if possible, can we minimize the impact of these causes?

Forms of minimization can include:

- **[prevent]** better learning material (documentation) if engineers are inadequately familiar with the modal μ-calculus formulae;
- **[guide]** a clearer or more verbosely descriptive (i.e. in words, not mathematics) syntax for specifications such that mistakes are less likely to be made, or are more easily spotted;
- **[constrain]** introducing an expert system like Remenska to replace the error-prone aspects with a more approachable process, or syntactically disallow 'obviously' wrong translations;
- **[correct]** an interpreter that could report on/visualize the meaning of the specification, such that engineers are supported in finding mistakes after they are made.

Research question 4 How could we transform either the syntax or the process in/by which engineers formalize requirements, such that mistakes are less likely to occur?

Research question 5 Is the new language accessible to engineers?

1.2 Main Contributions

We aim to motivate the earlier claims that the modal μ-calculus language is inherently difficult and exhibits the signs that we expect to find in a language that is difficult to learn and difficult to use. Additionally, we aim to find patterns in the mistakes that people make, such that we can achieve a better understanding of what exactly makes the language difficult to use, and how we might be able to improve upon these.

We can then use this information to design a more suitable language—one which preferably has the same expressive power as the modal μ-calculus formulae offer but is restricted and/or reworked in such a way that translations are more mechanical and intuitive, and promotes the use of structures close to our own intuitive interpretation of systems, such that human errors are less likely to play a role in the process, but not as restrictive as PSP and Remenska where a reported minimum of one in five requirements cannot be described at all.

Especially useful for extracting these mistake patterns are requirement documents with their translations to the modal μ-calculus. It would be beneficial if these documents were written by people who do not have much experience with writing modal μ-calculus formulae, so we can get a good idea of which concepts are difficult to learn. Additionally, this is arguably the most vital stage in the adoption of any language; a steep learning curve motivates the user to search for alternatives which are easier to learn, even if these alternatives are objectively less useful and/or powerful.

To source these requirement documents, we employ the lectures of Jan Friso Groote; he lectures a course on system validation (course code 2IMF30) at the Eindhoven University of Technology, explaining how mCRL2 works. The course consists of 7 weeks of in-depth lectures on the theory behind mCRL2, and focuses specifically on the syntax and semantics of modal μ-calculus formulae for multiple weeks. We assume that this is at least as good as the tutoring that inexperienced (in mCRL2) employees would receive in a business environment that would want to trial system validation via mCRL2.

However, we cannot claim that the comparison is perfect; designers in the industry will likely have a better grasp on formulating requirements, and creating consistent and encompassing requirements documents than the students of Jan Friso Groote. As such, we try to limit our conclusions to only the knowledge derived from the complexity and correctness of translations of requirements, not so much the structure and classes of the requirements itself. Regardless, we can still use requirements as defined by the students as a basis upon which to rate the ease with which the modal μ-calculus formulae are learned and used, based on the assumption that students are less likely to formulate similar requirements in similar ways (due to a lack of formal training) as long as they feel confident in their mastery of the language, and thus are more likely to construct similar structures if these were specifically taught to them and they do not feel confident enough to deviate from these.

In groups of two to three, the students are given the task to, over the course of 7 weeks and with supervision of Jan Friso Groote, pick a suitably complex system they want to validate, create at least 10 informal (i.e. natural language) requirements for this system, design an architecture consisting of at least three parallel actors, and model and verify these in the mCRL2 language using the formal (i.e. modal μ-calculus formulae) translation of the informal requirements.

We analyze the following data points;

1. The language structures (templates) used to formulate requirements, in order to map the diversity of requirements and evaluate to some degree the expressive power of modal μ-calculus formulae;
2. The frequency with which these language structures are used, in order to normalize the values for data points 3 and 4;
3. the frequency with which language structures are mistranslated, in order to find structures that are too complex to trivially translate into modal μ-calculus;
4. the possibility of each requirement being able to be translated into each proposed alternative, in order to evaluate the expressiveness of the alternatives.

Based on these values, we can determine not only whether modal μ-calculus is actually adequate, but also give a quantitative argument for the level of appropriateness of proposed alternative languages. Using these markers, we can then determine where the shortcomings and strengths of each language lies, such that we can use this as a starting position when trying to design a new language which is, in this context, better suited than either modal μ-calculus formulae or the alternatives offered by PSP and Remenska.

We will test the new language with students and professionals in the field of software engineering to evaluate their experience with using the language and make a comparison to their experience with the modal μ-calculus to argue the success of the project.

Finally, we provide a formal semantics for this new language, as well as a translation of this new language to mCRL2's modal μ-calculus, and prove that the translation preserves semantics.

1.3 Thesis outline

The rest of this thesis is organized as follows. In Chapter 2 (Context) we will go deeper into the context of the project and discuss why we assume that formal verification is so difficult. Chapter 3 (Background) will provide some background knowledge of the mCRL2 toolset, and the alternative specification frameworks of PSP and Remenska. Chapter 4 (Issues) discusses how we determined the issues faced by students in verifying formal process models, and why the alternatives are (ir)relevant to the use cases studied. Chapter 5 (Exploration) will go into detail about our proposed language and its structure. In Chapter 6 (Evaluation) we discuss the methodology of the user experience evaluation as well as its results and feedback. In Chapter 7 (Discussion) we discuss the implications of the project, and how we may interpret the results. Specifically, we discuss the threats to validity as a result of the assumptions made during the project. Finally, in Chapter 8 (Conclusions), we discuss the results of the thesis, and future directions.

Chapter 2

Context

mCRL2 is a formal specification language specifically designed to mathematically and rigorously model a software or physical process; it is used in conjunction with μ-calculus formulae to prove properties of these models, and therefore of the software or physical process it models. With it comes a broad range of tooling which is used in the verification and analysis of such models. Its applicability is broad, and companies have started adopting the practice of formally testing their software systems using it, often through higher level software packages that use mCRL2 as a compilation target; notable instances include CERN [11][12], ASML [13][14], and TNO Automotive [15].

Regardless of its broad applicability, the rate at which the language is adopted by users is quite low. One of the reasons for this, is that the entry cost of the language is incredibly high; the combination of mCRL2 and modal μ-calculus formulae (see section 3.2) is one that is mainly mathematical in nature, and little to no effort has been taken to make these languages accessible to people not familiar with the theories behind them.

mCRL2 has the benefit that most of its syntax can be trivially translated to concepts that most engineers are already familiar with; it looks similar to functional languages with some peculiarities, like unconditional choice and the notation of sequential behavior, that could be learned rather quickly even without proper understanding of the theory.

In contrast, the modal μ-calculus formulae do not benefit similarly; these formulae have no similarity to anything an engineer is reasonably expected to have experience with, the formulae do not have a structure that is easily interpreted by humans, and are difficult to translate to and from natural language. As such, learning this language is an expensive endeavor.

Engineers at ASML have conveyed a disapproval of the modal μ-calculus formulae used in mCRL2 models, because they are deemed to be too complex and too error-prone to serve as a scalable solution for their system validation needs. Simply said, the translation of requirements to formulae is not mechanical enough, and as a result too many human errors are introduced during this manual step. As such, there is a clear and strong need for a language that can be more intuitively understood, and is more straightforward in its translation.

However, we might question whether a new language is necessary in the first place. There are languages other than modal μ-calculus formulae that attempt to solve the same problem. One such system that is cited often and uses as motivation many of the points raised here is PSP (see section 3.3), created in response to the observation that the amount of expert knowledge needed to work with LTL logics would form a substantial obstacle to adoption rates [8]. PSP is originally defined in terms of LTL, but third-party translations to modal μ-calculus formulae can be found [16]. In addition to this, Remenska (see section 3.4) extends the PSP framework to be more expressive, and provides a computer-assisted process for non-expert users to generate the required modal μ-calculus formula by means of a question tree—essentially a decision tree-like questionnaire that progressively elicits the specific intention from the user.

Both these frameworks have an inherent issue that is difficult to work around; they are incredibly restrictive in their expressiveness. This is not necessarily bad; indeed, both PSP and Remenska are intentionally restrictive such that the choices are easier to comprehend and the framework can be familiarized more quickly. They can get away with this restrictiveness because they conclude that most requirements can be expressed using a small set of formulations; PSP can allegedly be used on around 70% of requirements, and Remenska claims an additional 10% improvement [10].

This has severe practical implications, however. If only a percentage of the requirements can be described using a framework, then that would necessitate the use of at least one other framework to describe the remaining requirements, which would in turn require additional training. Or in the absence of another framework, the user may try to fit the requirement into the original framework that does not support it, or abandon checking the requirement in the first place. As such, there is a very real consideration to be made as to whether the use of PSP or Remenska is appropriate for a given project.

Chapter 3

Background

To get a better idea of this report's goals and findings, it may be illustrative to give a brief overview of how the μ-calculus, PSP, and Remenska frameworks work and how they can be used. This section is dedicated to providing exactly that.

3.1 mCRL2

In the context of mCRL2, μ-calculus formulae are used to verify properties of an mCRL2 model. Foregoing too many intricacies, mCRL2 models may be best understood to define three things about the system that it models; the data types that the system uses internally, the actions—both external (i.e., interactions "visible" from outside the system) and internal (i.e., (mostly) communications of system components)—that the system can perform, and a formal description of which actions may be performed at a given point in time.

This latter component, the formal description of when actions may happen, usually comes in the form of an entity that tracks a certain state, and which (dis)allows actions based on its state. In the case of a (particularly dumb) coffee machine, for example, we might define a model that can receive a request for coffee for either a small or large cup, to which it will respond by activating a water pump, the heating element, a timer, and deactivates the water pump and heating element once the timer runs out, after which it will wait for a request again. In code, this would look as follows;

```
sort Size = struct small | large;  % defines a data sort named Size,
                                   %   of which ‘small‘ and ‘large‘ are instances

act
  get_request : Size;              % get_request has a parameter of type Size
  activate_pump;
  activate_heater;
  start_timer : Size;              % start_timer has a parameter of type Size
  timer_ended;
  deactivate_heater;
  deactivate_pump;

proc CoffeeMachine =              % this process models our system behaviour
  sum s:Size . (                  % instance the following sequence for all sizes
    get_request(s).               % the system will start a procedure only after
    activate_pump.                %   getting a request with a Size instance as
    activate_heater.              %   parameter.
    start_timer(s).               % the start_timer action uses the same parameter
    timer_ended.                  %   as the original get_request to ensure a
    deactivate_heater.            %   proper amount of coffee is dispensed.
    deactivate_pump
  ) . CoffeeMachine;

init CoffeeMachine;
```

This system tracks state implicitly; after an action `get_request` with a certain parameter `s` of type `Size`, the following actions must be done in order. Alternatively, we can define the same system as follows, with slightly more explicit state;

```
sort Size = struct small | large;

act get_request : Size;
  activate_pump;
  activate_heater;
  start_timer : Size;
  timer_ended;
  deactivate_pump;
  deactivate_heater;

proc CoffeeMachine(serving:Bool = false, size:Size = small) =
  (!serving) -> sum s:Size . (
    % if the system is not serving anything, listen for a request of a certain
    %   size, and change the state to reflect a pending request of size ‘s‘.

    get_request(s).
    CoffeeMachine(true, s)
  ) <> (
    % otherwise, if the system is pending service, fulfill the request of
    %   size ‘size‘, and return to the state in which we are waiting for a
    %   request (serving = false).

    activate_pump.
    activate_heater.
    start_timer(size).
    timer_ended.
    deactivate_heater.
    deactivate_pump.
    CoffeeMachine(false, size)
  );
```

```

init CoffeeMachine(false, small);
```

Behind the scenes, these definitions are translated to a labeled transition system. That is, the specific state that a system is in has no inherent meaning, other than what actions it allows to be performed at that state. We can informally motivate this as follows; if we cannot distinguish it from a correct system by all of the externally visible actions it performs, then it *is* a correct system.

It is important to note that the actions have a specific name that distinguishes it from other actions. Additionally, actions may have parameters, such as the `get_request` and `start_timer` actions in the previous examples. These can be used to make the life of the engineer easier by allowing to dynamically refer to actions (say, starting a timer of a given length), but also to handle parameters of infinite or even partially defined domains. In essence, the task of a parameter is to distinguish the action from other similarly named actions with different parameters. Indeed, if we were to replace the actions `get_request:Size` and `start_timer:Size` with the actions `get_small_request`, `get_large_request`, `start_small_timer`, and `start_large_timer`, and rewrote the mCRL2 model to use choice rather than the variable-binding operator `sum`, then we would essentially have an indistinguishable model.

3.2 modal μ-calculus formulae

The modal μ-calculus formulae allow us to reason about what sequences of actions are possible. To do this, we use the so-called *signature* of the mCRL2 model. That is, the data sorts, data functions (not discussed in section 3.1), and actions defined in the mCRL2 model. As mentioned before, we do not care about the state; only about what actions can be done from that state. For this, we tend to use the structures $[\alpha]\,\phi$ and $\langle\alpha\rangle\,\phi$; these can loosely be translated to the $\forall$ and $\exists$ quantifiers respectively on the states that are reachable from a certain state;

$[\alpha]\,\phi$ is **true** in a state iff for all α actions possible from that state, ϕ holds after taking said α action. Specifically, $[\alpha]$ **false** denotes that after all α actions, **false** must hold, which can only be **true** if there are no such actions α possible from the current state. If $\phi =$ **true**, then this is trivially **true** regardless of state or α.

$\langle\alpha\rangle\,\phi$ is **true** in a state iff in that state at least one action α is possible after which ϕ holds. Specifically, $\langle\alpha\rangle$ **true** requires that an action α is possible from that state, whereas $\langle\alpha\rangle$ **false** is trivially **false**.

Since these structures are resolved to a Boolean value, i.e. **true** or **false**, we can chain them. These chains have the expected meanings;

$[\alpha_1]\,[\alpha_2]\,\phi$ is **true** in a state iff from that state, all possible sequences of first an α_1 action followed by an α_2 action must result in a state where ϕ holds.

$\langle\alpha_1\rangle\langle\alpha_2\rangle\,\phi$ is **true** in a state iff from that state there exists a sequence of first an α_1 action followed by an α_2 action which results in a state where ϕ holds.

$[\alpha_1]\langle\alpha_2\rangle\,\phi$ is **true** in a state iff from that state, all actions α_1 result in a state where an action α_2 is possible which results in a state where ϕ holds.

$\langle\alpha_1\rangle[\alpha_2]\,\phi$ is **true** in a state iff from that state there exists an action α_1 such that in the resulting state, all possible actions α_2 actions result in a state where ϕ holds.

For convenience, we can group homogeneous sequences of $[\cdot]$ and $\langle\cdot\rangle$ together, using the following rewrite rules;

$$[\alpha_1 \cdot \alpha_2]\,\phi = [\alpha_1]\,[\alpha_2]\,\phi \qquad\qquad \langle\alpha_1 \cdot \alpha_2\rangle\,\phi = \langle\alpha_1\rangle\langle\alpha_2\rangle\,\phi$$

So far, we have only discussed these structures with the understanding that α is a singular action. However, there are situations in which we do not particularly care about which action it is as long as some action is being done, or where we might want one of a set of actions to be done. For convenience, we allow α to be a set of actions, with **true** denoting the set of all possible actions, and **false** denoting the empty set of actions. This allows us to define properties such as the absence of deadlock (i.e., an action is possible) in a state;

$$\langle\textbf{true}\rangle\,\textbf{true}$$

or more easily define properties on our coffee machine from section 3.1;

$$[\texttt{get\_request(small)} \cup \texttt{get\_request(large)}]\,\langle\texttt{activate\_pump}\rangle\,\textbf{true}$$

Alternatively, we may use the $\forall\gamma\,.\,(\phi)$ and $\exists\gamma\,.\,(\phi)$ quantifiers to construct these sets;

$$\forall s{:}\texttt{Size}\,.\,([\texttt{get\_request}(s)]\,\langle\texttt{activate\_pump}\rangle\,\textbf{true})$$
$$[\exists s{:}\texttt{Size}\,.\,(\texttt{get\_request}(s))]\,\langle\texttt{activate\_pump}\rangle\,\textbf{true}$$

We refer to such constructs as an action formula. Actions and action formulae are interchangeable (in the modal μ-calculus, not mCRL2): the action formula describing a singular action is analogous to that action. The entire set of operations allowed in an action formula is described in detail in Appendix A.

Additionally, we want to be able to write that we *eventually* or *after any number of actions* expect something to happen. We use the star notation to signify a sequence of zero or more actions. The property of global absence of deadlock would look as follows;

$$[\textbf{true}^\star]\,\langle\textbf{true}\rangle\,\textbf{true}$$

However, we cannot rewrite this star notation to any of the structures that we have discussed so far. To do this succinctly, we need to introduce a new structure entirely; the μ and ν fixed points. These are quite mathematical in nature and difficult to define in a way that is easy to reason about, but we can give a decently simple intuition about how to interpret them; the μ and ν fixed points are essentially units of recursion.

The ν fixed point is usually used in situations where we expect the recursion to potentially happen forever, or; infinitely. It essentially resolves to **true** unless somewhere in the recursion it encounters some state in which its body resolves to **false**. To assert that the system does not deadlock before some action `finish`, we could use;

$$\nu X\,.\,\big([\overline{\texttt{finish}}]\,X \land \langle\textbf{true}\rangle\,\textbf{true}\big)$$

Here we use $\overline{\texttt{finish}}$ to represent the set complement of `finish`, i.e., all actions except `finish`. As such, the property above can be read as recursing further into every action that is not a `finish` action, and requiring that in each of the states that we recurse into at least one action is possible. Since we do not recurse into `finish` actions, we do not require the absence of deadlocks after any such actions. We can define the box-star notation as a ν fixed point;

$$[\alpha^\star]\,\phi = \nu X\,.\,(\phi \land [\alpha]\,X)$$

As such, the following structure also represents the absence of deadlock before an action `finish`;

$$[\overline{\texttt{finish}}^\star]\,\langle\textbf{true}\rangle\,\textbf{true}$$

However, this is not to suggest that the fixed points (both μ and ν) are not incredibly more powerful than just a rigorous definition for the star notation. The body of the fixed points can be a lot more complex than in this simple example, and can define an infinite amount more (classes of) properties than what is possible with any of the structures so far. These are generally achieved through embedding fixed points inside others in interesting ways. The intuition described here can still be used in these cases, but their exact meaning can get intricate rather quickly.

Alternatively, the μ fixed point is usually used in situations where we require the recursion to end at some point, or; finite. In addition to resolving to **false** if it encounters some state in which its body resolves to **false**, it also resolves to **false** if it has to recurse infinitely to find whether it is **false**. This is particularly useful if we require something to happen *inevitably*; up until now we could only require that something is *possible* to happen after any point, not that it *must* happen at some point. For example, to say that the system must inevitably deadlock, i.e., there are no infinite paths, one might use;

$$\mu X\,.\,([\textbf{true}]\,X)$$

We can now define the diamond-star notation as a μ fixed point, as well;

$$\langle \alpha^{\star} \rangle\,\phi = \mu X\,.\,(\phi \vee \langle \alpha \rangle\,X)$$

Having properly butchered the semantic meanings of these structures, we refer to Appendix A for a more rigorous and mathematical definition of the syntax, semantics, and identities of the μ-calculus formulae.

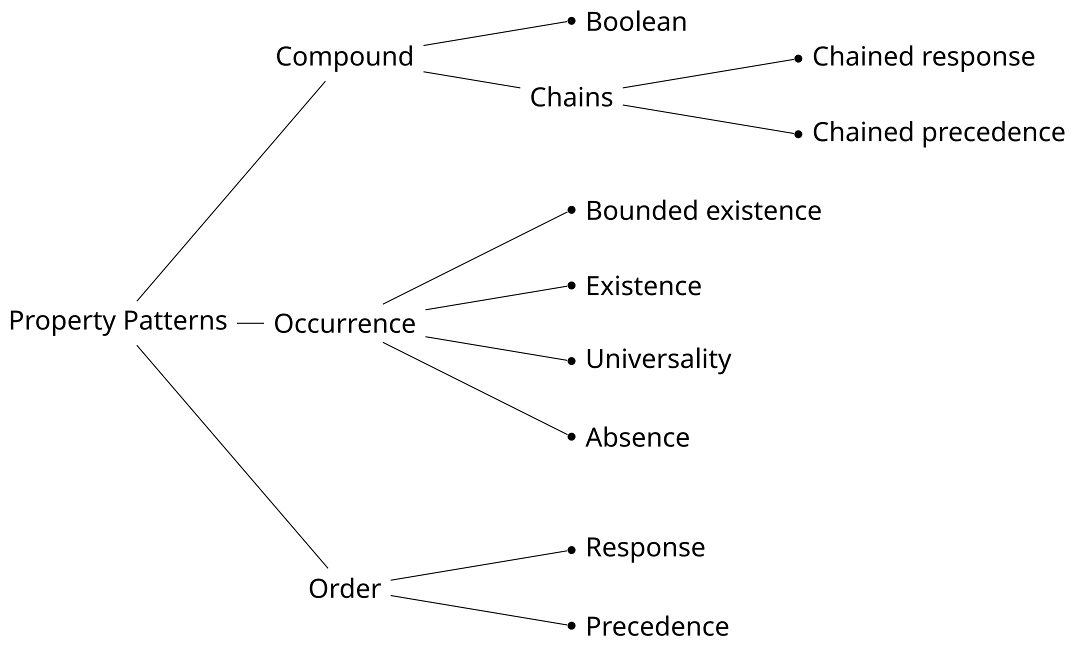
3.3 Property Specification Patterns (PSP)

The original paper by Dwyer et al. [8] asserts that actually working with property specification logics requires a decent amount of expert knowledge about the respective language one would be working in, even for relatively simple requirements. This in spite of tooling for property specification logics, such as LTL, CTL, and the modal μ-calculus, having received (at the time) a fair bit more interest and automated verification having matured a lot.

PSP is designed as an efficient framework with which to transfer experience from experts to people who have no reason to be more familiar with the respective language than strictly necessary to translate the requirements they want to express. The assumption is that the requirements that need to be expressed are relatively simple—or at least quite standard for the industry they are in—but the intricacies of the logic are such that, for someone to formulate the requirement correctly in said language, they need a high level of expertise.

As such, they propose a common knowledge base in which ubiquitous properties have been worked out by people with an expert knowledge in the language (specifically, of best practices), with detailed explanations and use-cases for each of the patterns, such that relatively lay people may simply "copy and paste" the respective components of the patterns they need to write what requirements they want. As such, we cannot truly speak of a singular PSP, as this essentially only describes the method by which such a knowledge base should be constructed. However, the original paper does provide an initial version upon which can be built. In the rest of this document we will use PSP to refer to the given knowledge bank rather than the framework.

PSP uses two levels of abstraction: patterns and scopes. Patterns describe properties, often resembling the explicit part of a natural language requirement. Scopes define the range of the program execution in which these patterns must hold, often (but far from always) resembling the implicit "intuitive, common sense" part of the natural language requirement. Using the requirement "when the throttle is released, no more power must be sent to the motors" introduced in the introduction, we can see the scope "after the throttle is released(, and before the throttle is applied again)" and the pattern "no (more) power must be sent to the motors". The patterns of PSP are defined in the following hierarchy;



With the following descriptions given for each;

Occurrence Patterns

Absence A given state/event does not occur within a scope. This pattern is also known as **Never**.

Existence A given state/event must occur within a scope. This pattern is also known as **Future** and **Eventuality**.

Universality A given state/event occurs throughout a scope. This pattern is also known as **Globally**, **Always**, and **Henceforth**.

Bounded Existence A given state/event must occur k times within a scope. Variants of this pattern specify at least k occurrences and at most k occurrences of a state/event.

Ordering Patterns

Precedence A state/event P must always be preceded by a state/event Q within a scope.

Response A state/event P must always be followed by a state/event Q within a scope. This pattern is also known as **Follows** and **Leads-to**. This pattern is a mixture of **Existence** and **Precedence**, and expresses a causal relationship between two subject patterns.

Compound Patterns

Chained Precedence A sequence of states/events $P_1, \ldots, P_n$ must always be preceded by a sequence of states/events $Q_1, \ldots, Q_n$. This pattern is a generalization of the **Precedence** pattern.

Chained Response A sequence of states/events $P_1, \dots, P_n$ must always be followed by a sequence of states/events $Q_1, \dots, Q_n$. This pattern is a generalization of the **Response** pattern. It can be used to express bounded FIFO relationships.

Boolean Combinations Most of the patterns delimit scopes and describe inter-scope properties in terms of individual events/states. There are cases where we want to generalize the patterns to allow for sets of states/events to describe scopes and properties. In some cases this is straightforward and disjunctions or conjunctions of state/event descriptions can be substituted into patterns; in other cases this yields the incorrect specification. These patterns outline how Boolean combinations can be applied in different cases.

Additionally, PSP presents 5 kinds of scope;

Global The entire program execution.

Before Q The execution up to a given state/event.

After Q The execution after a given state/event.

Between Q and R Any part of the execution from one given state/event to another given state/event.

After Q until R Like **Between Q and R** but the designated part of the execution continues even if the second state/event does not occur.

Note should be taken that the delimited scopes start at the very first instance of the "early/left" action, and ends at the first encountered instance of the "late/right" action. Note; for event-based verification, the scopes' intervals are open on both sides. So, for the scope **Between Q and R**, neither the delimiting **Q** nor the delimiting **R** action are included in the scope.

Each pair of scopes and patterns can be rewritten to mCRL2 such that they adhere to the descriptions given here (and the more rigorous definitions on the website by Dwyer et al. [17]). In fact, the CADP group has done this and made their translations public, over at [16]. However, these translations are not quite true to the original intentions of the definitions of Dwyer et al. Specifically, the right-delimiting actions may be deferred once if the property pattern requires the right-delimiting action to happen. Compare for "R is false before R" (**Absence Before Q**);

LTL (Dwyer et al.)	μ**-calculus formula** (CADP group)
$\Diamond R \rightarrow \neg R \,\mathcal{U}\, R$	$[\overline{R}^\star \cdot R \cdot \overline{R}^\star \cdot R]$ **false**
is trivially true, as no state satisfies R until the first state that satisfies R	false iff there exists a sequence containing two events (or; actions) R

This reinforces the claim made by Dwyer et al. that writing the right specification requires lots of expert knowledge and is sensitive to the specifics of the logic which may not immediately be obvious.

3.4 Remenska

Remenska's main goal was to create a tool with which to integrate model checking into the common software development cycle by automating the aspects that require formal methods expertise [9][10], but to achieve this goal a suitable formalism needed to be found. This formalism ended up being an extension to the PSP specification mentioned in section 3.3. Specifically, these extensions include the following scopes;

Until R Like the original **Before R**, but conditional on the **Global Existence** of R.

After Last Q Like the original **After Q**, but additionally requires that no more actions Q occur after the delimiting action Q.

Between Last Q And R Like the original **Between Q and R**, but additionally requires that no more actions Q occur after the delimiting action Q and before the delimiting action R.

After Last Q Until R Like the original **After Q Until R**, but additionally requires that no more actions Q occur after the delimiting action Q and before a possible delimiting action R.

As well as the following property patterns;

Order

- **Precedence Variant** Like the original **Precedence**, but the preceding action is required to happen.
- **Response Variant** Like the original **Response**, but the stimulating action is required to happen.

Compound

- **Precedence 3-Chain 1** Instead of allowing an arbitrary number of actions in P and Q, this handles the case of 1 action P, and 2 actions Q_1 and Q_2.
- **Precedence 3-Chain 2** Instead of allowing an arbitrary number of actions in P and Q, this handles the case of 2 actions P_1 and P_2, and 1 action Q.
- **Response 3-Chain 1** Instead of allowing an arbitrary number of actions in P and Q, this handles the case of 1 action P, and 2 actions Q_1 and Q_2.
- **Response 3-Chain 2** Instead of allowing an arbitrary number of actions in P and Q, this handles the case of 2 actions P_1 and P_2, and 1 action Q.
- **Constrained Response 3-Chain 2** Like **Response 3-Chain 2**, but defines an extra action R that, if encountered before some P_2, cancels the search for P_2—and thus the requirement for Q to happen after a P_2 does occur is void as well.

Note; in the thesis there are references to property patterns **Always Enabled**, **Existence Under Fairness**, **Precedence Variant Under Fairness** and **Response Variant Under Fairness**, but these are not actually implemented [18]. Additionally, the property pattern **Constrained Response 3-Chain 2** is implemented, but not mentioned in the thesis. Finally, since the μ-calculus formulae are based on those provided by the CADP group, these suffer the same issues as noted for their PSP translation in section 3.3.

Chapter 4

Issues

As mentioned in the introduction, we aim to prove that the μ-calculus formulae are difficult to learn and difficult to use. To substantiate these claims, we have sourced a number of requirement documents made by students of Jan Friso Groote's course on System Validation, where students learn to work with both mCRL2 and μ-calculus formulae. This is a first-year Master course, such that students should already be familiar with creating thorough requirement documents. The students are free to choose a system of their choosing to validate, but are given a predefined system as a fallback. As such, there are a few systems for which there exist a great number of requirement documents and thus there is a decent chance that most relevant requirements are stated, often in multiple forms; but there are also a decent number of requirement documents that describe a wide variety of systems. Specifically, these students are instructed to;

1. find a suitable software system to formalize (or use the provided suggestion)
2. write at least ten requirements in English
3. make a list of externally measurable actions (not artifacts produced by or for any software)
4. rewrite each requirement using a structured English grammar in terms of the externally measurable actions or the state of the system as a deterministic result of past actions.
5. rewrite all requirements in structured English to formal μ-calculus formulae
6. create the mCRL2 model
7. verify the model with the μ-calculus formulae

Here we define structured English to be the logic equivalent of what pseudo-code is to programming.

Most relevant to us is step 5; analysing the translation allows us to find how often these students make mistakes, what kind of mistakes they make, and whether there are any distinguishable patterns in *how* they go about translating these requirements. This should give us insights into how difficult the language is to use, since these students should be reasonably experienced with the μ-calculus formulae. Additionally, by assessing whether there are patterns in how the requirements are translated, we may find out how difficult the language is to learn by seeing how expressive they can be, or whether they fall into repetition by using the same technique over and over again.

Information is gathered in multiple stages. First, we capture template structures (that is; oft-occurring sentence structures) of the structured English requirements. We try to be as strict as possible when considering whether a new requirement should be classified as an earlier template; even if a requirement could have its wording slightly adapted to fit an earlier template, we should consider this requirement as a new template precisely because the wording is different. A concrete example of this is as follows;

- When the throttle is released, no more power must be sent to the motors
  `If <action> is performed, <object> must go to <state>`
- When the brake pedal is released, the brakes must be set to off
  `When <action1> is performed, <action2> must follow`
- The support vehicle can put the car in safe mode at any given time by means of a safe mode request
  `At any time when <command> is sent, <object> must go to <state>`

Even though the implementation may be identical, and the linguistic difference is only slight, it is still interesting to describe these as separate templates, because the ways one might go about translating them can differ drastically, and it allows us to find out whether different sentences with the same meaning are translated with human errors at different rates.

Secondly, we determine for each template whether it can be translated to PSP and Remenska. If it can be translated, we additionally determine with which scope and pattern (the taxonomy by which both PSP and Remenska classify their templates) we can describe these templates.

And finally, we determine for each requirement whether it was translated correctly. Since these requirements are written in English, it is not always perfectly clear what exactly is meant by the natural language description nor the structured English version. This is further aggravated by the possibility that the students do not have enough experience with writing requirements to appropriately express their intentions, or misuse jargon.

We additionally classify these mistakes in groups. This allows us to give a rate at which different classes of requirements are mistranslated, for how much of the total number of mistakes each class of requirement accounts for, and the most often made mistake for each class of requirement. Each class of mistakes is counted separately, and with these counts we can determine which classes of mistakes happen more frequently than others.

4.1 Common English requirement structures

These structures are manually extracted from the sourced requirement documents. There are a couple of drawbacks to this manual extraction; while attempts were made to keep the explicit templates as close to the original language in the requirements documents, some liberties are taken with rewording and restructuring the descriptions in order to achieve a more general wording. When descriptions are sufficiently different, but could be described by the same formal language, we still assign a separate structure to such a requirement. However, this is subjective and interpretations may differ.

One clear example of this phenomenon is the following requirement;

> `when a charging battery pack is full, the battery pack must be disconnected from charging.`

We have three structures that seem to fit this requirement, depending on interpretation of the language used;

1. `When <action1> is performed, <action2> must follow`
2. `When <object> is in <state1>, then it is also in <state2>`
3. `When <object> is in <state>, <action> must follow`

In this case, it is probably more appropriate to use the first structure, because disconnecting the battery pack from charging is clearly a *reaction* to a charging battery pack *becoming* full, but based purely on the language (and not any implicit models we may conjure) any of the three requirements seems to be valid. However, only two of these structures will return true if implemented; in the second version, we require that the battery is never simultaneously charging and full. But this would necessitate that the model either never fully charges the battery, or stops charging simultaneously with the system “finding out” that the battery is full. Neither of these is likely to represent the actual implementation (nor the intent of the requirement). Even worse, picking the second variant would result in an internally contradicting requirement;

> When <the battery> is ~~in~~ <charging and full>, then it is also ~~in~~ <not charging>

And therefore be more succinctly described as;

> While <the battery is charging>, <the battery> must never ~~perform~~ <report being full>

Which is the closest appropriate structure found in the sourced documents. The complete list of structures can be found in table 4.1.

The table clearly shows that there are only a few structures that are used significantly more often than others. In fact, the 5 (25%) most used structures account for approximately 50% of the requirements. If we combine the counts for IDs 3, 6, and 10, which are essentially the same structures with

ID	English structure	count
1	`While <state>, <object> must never perform <action>`	13
2	`<action> will only be done under <state>`	8
3	`When <action1> is performed, <action2> must follow`	16
4	`<object> will never perform <action> spontaneously (without being preempted by <action2>)`	5
5	`When <action1> is performed, <action2> and <action3> must follow (in order)`	6
6	`If <action> is performed, <object> must go to <state>`	4
7	`When <object> is in <state>, <action> must follow`	13
8	`<action> must always be possible while <state>`	5
9	`When <request> is sent, <response> must be received`	5
10	`At any time when <command> is sent, <object> must go to <state>`	3
11	`When <request> is sent, <response1> and <response2> must be received`	3
12	`When <object> is in <state>, it cannot perform <action1> before <action2>`	16
13	`<object> may only be in <state> if preceded by <action>`	2
14	`If <action> is not performed, then <object> is not <state>`	1
15	`When in <state1> and <action> is performed, go to <state2>`	9
16	`When <object> is in <state1>, then it is also in <state2>`	3
17	`When <object> is in <state>, then when <request> is sent, <response> must follow`	5
18	`When <object> is in <state>, then when <request> is sent, <response> must not follow`	4
19	`When <command1> or <command2> is sent, <action> must be performed`	1
20	`There is no deadlock`	1

Table 4.1: English language structures

different wordings, then this structure accounts for 10% of the requirements by itself. Additionally, if we assume that we can exit a certain state via only a single action, then we may claim that IDs 1 and 12 are similar, for a combined count of 21% of the requirements. Moreover, if we can generalize structures to account for multiple responses and an optional scope (or; state), then we might combine IDs 3, 5, 6, 7, 9, 10, 11, 17 and 19, for a combined count of approximately 41%.

These last two numbers are highly disingenuous; while the language is similar to a degree, the specific implementation needed is significantly different between each of these structures, even though this may not be immediately obvious when using trivial examples. The necessary assumptions seem rather simple, but have a big impact on the resulting μ-calculus formulae.

However, this does give us a good insight in the type of requirements that are being written; most concern themselves with something not happening before something else, or something necessarily happening after something else. Interestingly enough, these two concepts are close to two template μ-calculus formulae given in the System Validation course material; the safety property (4.1) and the liveness property (4.2);

$$[\textbf{true}^\star \cdot a \cdot \overline{b}^\star \cdot c]\ \textbf{false} \tag{4.1}$$

$$[\textbf{true}^\star \cdot a]\ \mu X\ .\ \big([\bar{b}]\ X \wedge \langle\textbf{true}\rangle\ \textbf{true}\big) \tag{4.2}$$

This would suggest that either the μ-calculus is not expressive enough to deviate from these templates, that the students are not confident enough to deviate from these templates, or that it is simply not all that interesting to deviate from these templates.

The first possibility is quite easily proven false; there are requirements that do not fit into these classes and are still being translated to μ-calculus formulae. The other two possibilities are more difficult to determine the validity of, which will be discussed later.

4.2 Common mistakes

By analyzing the students' translations of natural English requirements to structured English requirements, it is easy to see that by far the most mistakes in translation are made because of complex logic structures that consists of multiple components. Notably, this means that the requirements are not atomic; a property which is widely accepted by the industry to be best-practice in designing requirements. While students are often aware enough of this best-practice that they avoid the obvious structures where the keywords "and" and "or" are used, structures like "if and only if" are often only translated as if using an "if" structure;

English requirement	**Structured requirement**
If and only if solar power is available, power is supplied from the solar panel to the battery	Before every `power(solar, battery)` action, the last preceding `solar_panel(s)` action must have `s = true`

Other times, it seems as if students will translate a requirement to a similar seeming template structure which is taught in the lectures, but not actually equivalent to the requirement they want to prove;

English requirement	**Structured requirement**
The crew must be able to put the car into safe mode at all times	After any `panic_button` action, a `safe_mode` action must eventually occur

However, while these mistakes point to a larger issue at hand, and it would be interesting to see if these issues pop up with subjects who are more experienced with writing requirement documents, they are not quite as interesting to the aim of this report; we have already assumed that the students are not well versed in writing requirements, which includes being clear and unambiguous in their communication.

But when considering the translations from structured English requirements to μ-calculus formulae, the same sloppiness presents itself; over 42% of requirements is mistranslated with comparatively trivial mistakes;

Structured requirement

When *read_brake_pedal(ON)* occurs and brake 1 and brake 2 are not **defect**, actions *brake1(ON)* and *brake2(ON)* will follow.
When *read_brake_pedal(ON)* occurs and brake 3 and brake 4 are not **defect**, actions *brake3(ON)* and *brake4(ON)* will follow.

μ-calculus

```
nu X(s:Press=ON) . (
    [read_brake_pedal(ON) .
     brake_monitor_light1(OFF) .
     brake_monitor_light2(OFF) .
     brake1(s) . brake2(s)] X(OFF) ||
    [read_brake_pedal(ON) .
     brake_monitor_light3(OFF) .
     brake_monitor_light4(OFF) .
     brake3(s) . brake4(s)] X(OFF))
```

This is quite clearly a faulty understanding of how the ν fixed point works. In the 136 considered requirements, there are 10 other requirements that suffer from similar misinterpretations of the meaning or use of the ν fixed point, in total accounting for over 21% of the mistakes, and over 8%

of all requirements.

However, the most often made mistake is one that concerns a different form of poor understanding; how to handle inevitability. A lot of students are under the impression that the best way to describe that some action α must inevitably happen, is to use some structure similar to $[\textbf{true}^\star]\langle\textbf{true}^\star \cdot \alpha\rangle\ \textbf{true}$. But this merely describes that at any point there exists a future where the action α is possible. The difference is small, but distinct; if we consider the state machine in figure 4.1, we can see that the α action is always inevitably possible, but does not ever have to be done. Mistakes like these account for over 46% of mistakes, and over 17% of requirements have this mistake. The correct requirement is $[\textbf{true}^\star]\ \mu X\ .\ ([\overline{\alpha}]\ X \wedge \langle\textbf{true}\rangle\ \textbf{true})$, similar to (4.2).

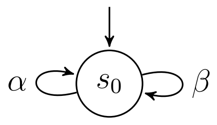


Figure 4.1: "Inevitably α" only applies under the fairness assumption; $[\textbf{true}^\star]\ \langle\textbf{true}^\star \cdot \alpha\rangle\ \textbf{true}$ does hold, whereas $[\textbf{true}^\star]\ \mu X\ .\ ([\overline{\alpha}]\ X \wedge \langle\textbf{true}\rangle\ \textbf{true})$ does not.

The complete list of mistake types can be found in table 4.2, and a breakdown of the amount and types of errors made per structured English requirement class can be found in table 4.3.

ID	Mistake type	count
1	Initial condition; requires action to reset the state, but this is technically not something we may assume without checking this property individually	1
2	Does not check for infinite loops or fairness	24
3	Requires that an initial action has been done, but is not checked	4
4	Initial condition; requires that a substate has been set to a specific instance, but it may have been set to other instances instead. i.e. partial state check	1
5	Wrong handling of state	3
6	Claims to some property must be met before an action can be done, but then requires the action to be done.	1
7	Incorrect/partial translation of conditions	1
8	wrong use of ν fixed point operator	11
9	error in logic	4
10	syntax error	3
11	safety instead of liveness, or vice versa	1
12	too complex/strong of a structure, tests requirement only partially	3
?	difficult to categorize	5

Table 4.2: Common mistake types

ID	count	Correctness	Accounts for	Top mistake	
1	13	92.31%	1.92%	7	1×
2	8	62.50%	5.77%	3	2×
3	16	31.25%	21.15%	2	8×
4	5	80.00%	1.92%	3	1×
5	6	66.67%	3.85%	2	1×
6	4	25.00%	5.77%	2	2×
7	13	30.77%	17.31%	2	4×
8	5	80.00%	1.92%	2	1×
9	5	20.00%	7.69%	2	3×
10	3	33.33%	3.85%	2	2×
11	3	66.67%	1.92%	8	1×
12	16	68.75%	9.62%	?	2×
13	2	50.00%	1.92%	8	1×
14	1	100.00%	0.00%	N/A	0×
15	9	55.56%	7.69%	2	2×
16	3	66.67%	1.92%	6	1×
17	5	100.00%	0.00%	N/A	0×
18	4	100.00%	0.00%	N/A	0×
19	1	100.00%	0.00%	N/A	0×
20	1	100.00%	0.00%	N/A	0×

Table 4.3: Mistake per structured English requirement analysis

4.3 Alternatives

When comparing the requirement classes to the alternatives PSP and Remenska, it becomes quickly obvious that neither PSP nor Remenska is suited for complex requirements. Specifically, requirements where states are involved are rather tricky to describe in the alternatives. This is not necessarily because there exist no parallels in PSP or Remenska; indeed, the Between-Q-and-R scope is likely devised specifically because of this reason. Where both alternatives fall short, however, is in the complexity that this structure would allow. Both frameworks constrain the "between" scope to single delimiting actions (one preceding, one succeeding the scope). Therefore, any more complex state cannot reasonably be implemented in either alternative.

To illustrate this difficulty, we will reuse the rechargeable battery pack that has featured in previous examples; this battery pack has three states of charge; it is either empty, charged, or full. We introduce 4 actions which represent changes in this state; `e2c` (from empty to charged), `c2e` (from charged to empty), `c2f` (from charged to full), and `f2c` (from full to charged). If we wanted to verify that a certain property—say, the car always starts in response to starting the car—holds while the battery is charged (but not full), then we cannot simply use the Between-Q-and-R scope, because this cannot fully describe all the delimiting actions between which we might find ourselves in the correct state. Indeed, we would need four such scopes in conjunction; Between-`e2c`-and-`c2e`, Between-`f2c`-and-`c2f`, Between-`f2c`-and-`c2e`, and Between-`e2c`-and-`c2f`. For each of these scopes, the pattern should be Response. (In fact, this is not entirely accurate either; remember that in the original definitions, the Between-Q-and-R scope always matches the first Q instance, whereas here we assume minimally matching = non-overlapping scopes. That is, we want the latest Q before a matching R).

This is also the single instance where Remenska lacks a feature that PSP offers (outside of not allowing more than three actions per chain property pattern); Boolean combinations of templates. However, Remenska technically only offers a tool that makes the mechanical translation of the question tree to the μ-calculus as an attempt at eliminating human error, but encourages human intervention to fine-tune the result. This is likely because the tool would be a lot more complex if Boolean combinations were added, whereas humans would likely be able to manually create these combinations without too much effort or error. However, regardless of whether it is possible within the framework, it should be clear that this solution is not scalable in any case; if we would instead model a cruise control mode which can be set to integer speeds of 0 to 150 km h^{-1}, and wanted to check properties regarding safety measures activating at and above 70 km h^{-1}, the resulting μ-calculus formula would get incredibly complex, and likely inadvertently introduce more possibilities for human error than the completely manual procedure would have.

Ignoring this shortcoming, and considering only the simplest cases for each structure, in table 4.4 we have mapped each structure to the appropriate PSP and Remenska classification where possible (and feasible, see previous paragraph). Notably, Remenska does not seem to give us any benefit over PSP. Additionally, 13% of requirements cannot be described by PSP, whereas 19% cannot be described by Remenska. Moreover, another 38% of requirements considers a state and can only be translated if the state were trivial (i.e. one incoming action, one outgoing action).

ID	count		PSP Scope	PSP Pattern	Rem. Scope	Rem. Pattern
1	13	██	Between-Q-and-R	Absence	Between-Q-and-R	Absence
2	8	█	Boolean combinations		not possible	
3	16	███	After-Q	Existence	After-Q	Existence
4	5	▌	Before-Q	Existence	Before-Q	Existence
5	6	▌	After-Q	Chain-response	After-Q	Chain-response
6	4	▌	After-Q	Existence	After-Q	Existence
7	13	██	Between-Q-and-R	Existence	Between-Q-and-R	Existence
8	5	▌	Between-Q-and-R	Universality	Between-Q-and-R	Universality
9	5	▌	Globally	Response	Globally	Response
10	3	▎	After-Q	Existence	After-Q	Existence
11	3	▎	Globally	Chain-response	Globally	Chain-response
12	16	███	Between-Q-and-R	Precedence	Between-Q-and-R	Precedence
13	2	▎	Before-Q	Absence	Before-Q	Absence
14	1	▏	Before-Q	Absence	Before-Q	Absence
15	9	█	not possible		not possible	
16	3	▎	not possible		not possible	
17	5	▌	Between-Q-and-R	Response	Between-Q-and-R	Response
18	4	▌	not possible		not possible	
19	1	▏	not feasible		not possible	
20	1	▏	not possible		not possible	

Table 4.4: Translation of the structured English requirement classes to alternative formalizations

Considering this, it should be clear that PSP or Remenska alone could never be sufficient as the sole solution to the problem. As predicted, they simply miss too large a portion of the requirements to stand alone. Simply adapting these frameworks to be more suited to this domain is not likely to work either; regardless of the type of state-handling scope we add, for example, there are many types that we will miss. Additionally, the structures that were impossible to translate are not quite similar enough to group together in any meaningful way, and as the number of uses of each of these structures is quite low, their inclusion will likely remove what positive aspects the frameworks had; a small size that was easy to comprehend and familiarize. As such, it is clear to us that the solution should be found elsewhere.

That is not to say that PSP nor Remenska are not useful, even in this context; it may be extremely useful to have a system with which a significant portion of the requirements may be translated rather quickly and with little to no expert knowledge necessary. However, the trade-off is that if they were used in conjunction with other systems, the parties responsible for translating requirements using PSP or Remenska would have to be knowledgeable in what requirements can be translated, and what the limits of these formalizations are. This brings an additional level of complexity to the process, as the user without expert knowledge would essentially be in charge of deciding which requirements would be translated using which formalization. Additionally, a secondary formalization would be necessary to fill the gaps, which would require additional expert knowledge.

Chapter 5

Exploration

5.1 Motivation

The μ++ language is primarily designed around the concept of state-based event handling. That is, the paradigm that the language is based on is that of requirements on future behavior and the state of the process after some initiating event.

This is motivated by our study of requirement documents written by students (Section 4); most of the requirements are formulated from the perspective of either some condition on the state of a process being met, or some initiating event. Some popular requirement structures are:

- while <`state`>, <`object`> must never perform <`action`>,
- when <`action1`> is performed, <`action2`> must inevitably follow,
- when <`object`> is in <`state`>, <`action`> must inevitably follow,
- when <`object`> is in <`state`>, it cannot perform <`action1`> before <`action2`>,
- <`action`> may only be done while <`state`>.

It should be plain to see that all but one of these requirements use terms that are not native to the modal μ-calculus; there is no easy way to reason using states or from the perspective of certain (sub)processes (or; objects), i.e., ignoring irrelevant aspects such as other parallel processes. The prevalence of state and object-based reasonings in the studied requirement documents suggests that we should consider using a paradigm that supports them natively.

5.1.1 Keeping track of state

To achieve this, we use monitors to track the states of (sub)processes as intuitively as possible. These monitors have an internal state which represents the state of some (sub)process, as well as transitions based on which events (actions) are performed in the process. As a simple example;

```
monitor powered(OnOff state = off):
  on powerOn:  powered(state=on)
  on powerOff: powered(state=off)
```

This example demonstrates a simple monitor which tracks a certain object's powered state, i.e., whether it is turned on or off. The state of the monitor exists of a single variable called `state`. The `state` variable is of type `OnOff`, here taken to mean a Boolean set containing the literals on and `off`. Additionally, we can see that the monitor responds to two events; a `powerOn` event, and a `powerOff` event, upon which it sets its state to on and `off`, respectively. If we were to translate this monitor to the modal μ-calculus—which would not give a useful result, like how the monitor on its own is not useful: nothing is being checked—we would get the following requirement;

```
nu powered(state:OnOff = off) . (
  [powerOn]                powered(on)     &&
  [powerOff]               powered(off)    &&
  [!powerOn && !powerOff] powered(state)
)
```

We see that the modal μ-calculus translation is virtually identical to the μ++ version. Indeed, this is by design; μ++ is not meant to be a novel language but a restriction on the feature set of the modal μ-calculus. However, we can (and do) introduce quality-of-life improvements that make the verbose and tedious aspects of creating modal μ-calculus formulae easier. A good example of this is the case in which the state of the monitor does not change; in the modal μ-calculus we have to explicitly note the set of actions (events) which have no impact on the state, because the maximal fixed point ν can be used for more than just a state monitor, whereas in μ++ we restricted the syntax and can therefore infer which events have no effect on the state merely from its context (specifically, the complement of the set of events which *do* impact the state). As a result, in the simple examples presented earlier, the μ++ monitor does not need a fourth line to describe secondary behavior.

The `on` keyword allows an action formula af as defined for the modal μ-calculus. This would give the user the freedom to introduce conditionals and sets over data. However, at times we will have to define the same sets of data or conditionals for multiple actions, and thus it would be nice to allow a scope in which these are defined only once. Additionally, this would allow us to define dynamic 'reactions' as well:

```
monitor car(Int speed = 0, Bool legal = true):
  for n in Int:
    if n > 130:
      on speed(n): car(speed=n, legal=false)

    if n <= 130:
      on speed(n): car(speed=n)
```

It is not always the case that in the undefined behavior the state should stay equal. For example, if we wanted to keep track of when the powered state changed instead of its specific state at any point, we should “reset” the state on uninteresting events instead.

```
monitor button(Bool pressed = false):
  on button_press: button(pressed=true)
  otherwise:       button(pressed=false)
```

The `otherwise` clause allows us to define behavior for all events that statements in its current scope do not address. However, these should be used with caution, as they may not always do what one might expect:

```
monitor powered(OnOff state = off, Int toggles = 0):
  for state’ in OnOff:
    if (state’ != state):
      on power(state’): powered(state=state’, toggles=toggles + 1)
    otherwise:          powered(state=state’)                ← very, very wrong
```

In this example, we might erroneously expect that the `otherwise` clause would set the state of the monitor to the new state `state’` in the cases where we encounter an event `power(state’)` for which `state = state’` holds. This is only partially correct; it will additionally set the state to *both* on *and* off for all events other than `power(...)`.

It may seem weird how we can set the state to be both on and off, but we can intuitively understand this as follows; we create two recursions, each with their own state, but which both have to hold. We can use this property to create very interesting requirements. To illustrate, we can require that the number 42 does not have a simple bit representation as follows;

```
% mCRL2 model
act step, stop;
proc create_binary_number = step . ((stop . delta) + create_binary_number);
init create_binary_number;
```

```
monitor bit(Int value = 0):
  on step: bit(value = 2 ∗ value + 0)  % append 0
  on step: bit(value = 2 ∗ value + 1)  % append 1

require:
  after stop:
    assert bit.value != 42  % false
```

In this example, each action `step` essentially 'picks' a next bit where the specific choice is abstracted away from, and the monitor similarly abstracts over this choice by allowing both a 0 and a 1 bit to be picked. Since we can pick a sequence of 0s and 1s to represent 42 (namely 101010), the requirement that we should not be able to stop on a value of 42 resolves to false. This is in spite of the missing information in the model (due to abstraction).

However, that is more of a drawback than a good thing; because such structures are legal, they will not throw an error when verifying the requirement. As such, extreme care should be taken when defining the requirement that such properties are not introduced without intending to. Because this format is extremely unlikely to be the intended behavior when using `otherwise`, we forcefully disallow `otherwise` keywords inside `for` keywords. If for some reason this behavior is what the requirement calls for, then the behavior must be explicitly written down using an `on` keyword instead.

5.1.2 Composable monitors

In the interest of reusability, we may want to define multiple monitors that each have their own responsibility. This is especially useful when we want to verify more complicated requirements that have multiple components to them; the more complex we make individual monitors, the less convinced we can be that they are indeed defined correctly. There is, after all, no step that would verify that we defined our formula correctly.

In the same vein, we might want to extend another monitor to create a compound monitor. For example, if we wanted to keep track of whether some state is entered after some action `preceding`, we could create a general monitor to keep track of the state (which we could reuse for other requirements), and a separate monitor to keep track of the preceding action;

```
monitor object(State state = s0, Bool left = false):
  on state_in:  object(state=s1, left=false)
  on state_out: object(state=s0, left=true)
  otherwise:    object(          left=false)

monitor preceded(Bool b = false):
  on preceding: preceded(b=true)
  otherwise:    preceded(b=b && !object.left)
```

As you might be able to see, we can use the state of other monitors in a monitor. In this example, we use `object.left` to reset the state of `preceding` when the object leaves whatever state we are monitoring for. Attentive readers may have noticed, however, that the state of `preceded` does not behave exactly as we would want; since `object.left` only updates *after* the state is left, and `preceded.b` only updates *after* an event happens *and* `object.left` is true—essentially, `preceded.b` lags behind by one event.

For this use case, we propose a new unary prefix operator `>` which instead of using the current value of the following expression, it will use the values for the *next* state, instead. This operator only works in monitors, not in requirements. To be able to compute the proper value for the assignment, the definitions should not have circular dependencies. Using this new operator, we can define `preceded` in a way in which it does not lag behind;

```
monitor preceded(Bool b = false):
  on preceding: preceded(b=true)
  otherwise:    preceded(b=b && !>object.left)
```

Alternatively, we could offload the task of keeping track of whether the state is fresh to `preceded` too, as we can now do this without having to rely on knowledge outside of how the `preceded` monitor interface was defined (specifically, without knowledge of what events cause `object` to leave state `s1`);

```
monitor object(State state = s0)
```

```
  on s_in:  object(state=s1)
  on s_out: object(state=s0)

monitor preceded(Bool b = false):
  on preceding: preceded(b=true)
  otherwise:    preceded(b=(b && !(>object.state == s0 &&
                                   >object.state != object.state)))
```

5.1.3 Specifying a requirement

Writing a requirement is quite similar to writing a monitor, but instead of changing the state of the monitor, we assert boolean propositions. In addition to the keywords we could use for monitors, we can also use `initially` for assertions that should only be made in the initial state, and `invariant` for assertions that should be made in every reachable state. If we take `response`(af) to mean the proposition that at any time before an action af is taken there is a possibility of taking an action af in the future, then we might write the "no deadlock" requirement in the following way;

```
require:
  invariant:
    assert response(any)        ← deadlock-free
```

Here we use the special action formula any to denote the action formula matching any action. Because using `true` and `false` as action formulae may not be the most intuitive, we have aliased these keywords with any and `paradox` respectively.

Similarly to monitors, requirements can also respond to actions;

```
require:
  after request_shutdown:
    assert response(shutdown)   ← assuming fairness, shutdown follows a request_shutdown
```

Using monitors should hopefully be quite intuitive, then;

```
monitor object(State state = s0)
  on alarm:   object(s=s1)
  on recover: object(s=s0)

monitor preceded(Bool b = false):
  on emergency: preceded(b=true)
  otherwise:    preceded(b=(b && !(>object.state == s0 &&
                                   >object.state != object.state)))

require:
  invariant:
    assert object.s == s1 => preceded.b     ← an emergency event precedes the alarm state
```

Outside of boolean equations on the used monitors' states, we can use propositions on actions, too. We have introduced one such proposition keyword earlier; `response`. However, there is more to this proposition than we originally mentioned; the full signature of the `response` keyword is as follows;

```
response[*](                 ← optionally: allow avoidable or non-existent target actions
  [inevitably]               ← optionally: the response must unavoidably happen
  af                         ←             target actions
  [before af]                ← optionally: must not encounter any of these
```

```
    [unless af]            ← optionally: give up early after any of these are taken
    [before∗ b]            ← optionally: must never be true
    [unless∗ b]            ← optionally: give up early when this is true
  )
```

The order in which these are applied is roughly from bottom to top. That is, starting from the 'initial' state (that is, the one in which the response operator is asserted), the following rules are applied in order;

1. If the boolean expression in the unless∗ clause is true, the result of this state is true.
2. If the boolean expression in the before∗ clause is true, the result of this state is false.
3. If an action described by the before clause (not contained in the unless clause) can be taken, the result of this state is false.
4. If the response operator is not starred and there does not exist a path such that an action described by the unnamed clause can eventually be taken, the result of this state is false.
5. If actions can be taken which are in neither the unless, before, nor the unnamed clause, the result is the conjunction ($\wedge$) of the results of all states reachable through these actions. If this conjunction cannot be resolved due to cyclic dependencies, then the result of this state is false if the inevitably keyword is used, true otherwise.
6. By default, the result of this state is true.

Note that some cyclic dependencies can be resolved; since each of the states in the cycle have as a result the conjunction ($\wedge$) of results of states reachable by 'uninteresting' actions, if any of these reachable states resolves to false, this means that the conjunction is necessarily false regardless of the unresolved value of reachable states in the cycle. As such, states in reachable cycles in a check without an inevitably clause may also resolve to false!

The reasoning behind this behavior of the inevitably clause is as follows: if the cycle is unresolvable, that means that a **true** result is inescapable. If a **false** result was reachable from the cycle, then clearly the cycle should resolve to **false** as well. However, inescapable does not mean that it must inevitably happen. Indeed, we can repeatedly traverse the cycle to delay the **true** result indefinitely.

response is more useful than it might initially seem. To demonstrate the power of this proposition, we consider the following two requirements;

1. When data has been received using read(d) for some $d, d', d'' \in D$, then—provided that no more actions read(d') are received—the first following action send(d'') must satisfy $d = d''$.
2. When a request_shutdown event happens, the machine should emit events flush_journal and shutdown in that order, unless an event cancel_shutdown happens at any point before shutdown is emitted.

We can formalize either requirement using monitors, but it is not at all necessary to go that way. Since the first requirement does not require the 'correct' send action to happen—merely that *if* one happens, it has the correct data—we can use `response*`;

```
require:
  for d in Data:                                    % for some d in Data
    after read(d):                                  % when data has been received
      assert response*(                             %   using read(d)
                 exists d'':D . send(d'')           % first following send(d'')
        before exists d'':D . send(d'') && val(d != d'') % must satisfy d=d''
        unless exists d' :D . read(d' )             % provided no more read(d')
      )                                             %   are received
```

Notice here that the `before` clause takes precedence over the unnamed clause. Considering only these two clauses, we can slightly reword and simplify the rules given above: “If an action described by the `before` clause can be taken before an action described by the unnamed clause is taken, the result is false”. Assuming $d'' \neq d$, this further simplifies to the following; “If a send(d'') can be taken before a send(d'') is taken, the result is false”. Using this verbiage, it should be clear that the result is false if $d'' \neq d$.

Compare this to the same requirement using a monitor, where the reasoning is a lot more difficult to understand;

```
monitor last(Data d = null, Bool check = false):
  for d' in Data:
    on read(d'): last(d=d', check=true)
    on send(d'): last(      check=false)

require:
  for d in data:
    if last.check && d != last.d:
      after send(d):
        assert false
```

For the second requirements, we need to test that a proposition `inevitably` follows. On its own, this keyword asserts that its contents must unavoidably happen in a finite number of steps. Inside a `response` keyword it has a similar meaning; the check must always end in a finite number of steps (get the response, or trigger an `unless` clause). Alternatively, we might assume that the model is fair.

```
require:
  after request_shutdown:
    assert response(inevitably flush_journal before shutdown unless cancel_shutdown)
    assert response(inevitably shutdown                      unless cancel_shutdown)
```

Here, the difference with the version using a monitor is even more pronounced;

```
monitor observed(Bool flush = false, Bool cancel = false, Bool shtdwn = false):
  on flush_journal:    observed(flush=true)
  on cancel_request:   observed(             cancel=true)
  on shutdown:         observed(                           shtdwn=true)
  on request_shutdown: observed(flush=false, cancel=false, shtdwn=false)

require:
  after request_shutdown:
    assert inevitably(observed.cancel || observed.flush)
    assert inevitably(observed.cancel || observed.shtdwn)

  if !observed.flush:
    after shutdown:
      assert false
```

To improve the usability of `response` a little bit in relation to sequences of actions, we introduce the `sequentially` keyword, which will translate to a number of `response` assertions. Rewriting the second requirement to use `sequentially` would produce;

```
require:
  after request_shutdown:
    assert sequentially [
      inevitably flush_journal unless cancel_shutdown,
      inevitably shutdown      unless cancel_shutdown
    ]
```

Notice that we do not have to write `before shutdown` in this instance; the `sequentially` keyword takes care of this for us by implicitly adding each of the later unnamed clauses (and only the unnamed clauses) to an entry's `before` clause.

We can use a separate `inevitably` keyword to require that a certain condition will eventually be met. The values of the monitor variables used in this `inevitably` expression will be updated during its evaluation. That is, they do not retain their values of the current state. To illustrate, we translate the requirement "if we request a machine to turn on, it must eventually be turned on";

```
monitor machine(Status power = off):
  for state in Status:
    on changeMachineStatus(state): machine(power=state)

require:
  after receiveMachineStatusRequest(on):
    assert inevitably(machine.power == on)
```

We can combine the properties that we have encountered so far using Boolean operations to create quite complex requirements. Introducing a final few keywords; `possible(`R`, b)` requires that from this state, a sequence R is possible such that b holds afterwards. `afterall(`R`, b)` requires that after all possible sequences R, b must hold. Note; `!possible(a, b)` $=$ `afterall(a, !b)`. `mcf(...)` allows a modal μ-calculus formula, for those rare few cases where μ++ is inadequate.

5.2 Syntax of μ++

A μ++ formula contains two types of structures; monitors and requirements. Typically, we would only need a single requirement as multiple requirements should combine trivially. However, when creating a monolithic μ++ formula, it may be useful to individually name and verify these requirements to help identify issues during verification.

```
           μ++ ::= μ++*
                 | 'require' [identifier] ':' requirement+
                 | 'monitor' identifier '(' data_def [',' data_def ]* ')' ':'
                   monitor

    identifier ::= /([A-z]|_[A-z0-9'])(_?[A-z0-9'])*_?/
        number ::= /(0|[1-9][0-9]*)\.?(e[+-]?[0-9]+)?/
                 | /(0|[1-9][0-9]*)?\.[0-9]+(e[+-]?[0-9]+)?/
       comment ::= '%' /[^\n\r]*/

          sort ::= identifier
          data ::= identifier 'in' sort
      data_def ::= sort identifier '=' data_instance
 data_instance ::= identifier ['.' identifier]
                 | t
        assign ::= identifier '=' data_expression

 monitor_scope ::= 'if' boolean_expression ':' monitor
                 | 'for' data [',' data]* ':' monitor
                 | 'on' action_formula ':' identifier '(' assign [',' assign]* ')'
       monitor ::= monitor_scope+
                   ['otherwise' ':' identifier '(' assign [',' assign]* ')']

   requirement ::= 'if' boolean_expression ':' requirement+
                 | 'for' data [',' data]* ':' requirement+
                 | 'after' action_formula ':' assertion+
                 | 'initially' ':' assertion+
                 | 'invariant' ':' assertion+

     assertion ::= 'if' boolean_expression ':' assertion+
                 | 'for' data [',' data]* ':' assertion+
                 | 'assert' proposition
```

These conclude the structural components of the μ++ language. Using these definitions, we can describe exactly *when* (or; *from what point in time*) we want to verify (or; `assert`) certain properties. What follows are the production rules that make up propositions.

$$\begin{aligned}
\text{proposition} ::= {}& \text{boolean\_expression}[b|\text{proposition}/b] \\
|\ & \texttt{'response'}\ [\texttt{'*'}]\ \texttt{'('}\ \text{response}\ \texttt{')'} \\
|\ & \texttt{'sequentially'}\ \texttt{'['}\ \text{response}\ [\texttt{','}\ \text{response}]^{\star}\ \texttt{']'} \\
|\ & \texttt{'inevitably'}\ \texttt{'('}\ \text{proposition}\ \texttt{')'} \\
|\ & \texttt{'possible'}\ \texttt{'('}\ \text{regular\_af}\ [\texttt{','}\ \text{proposition}]\ \texttt{')'} \\
|\ & \texttt{'afterall'}\ \texttt{'('}\ \text{regular\_af}\ \texttt{','}\ \text{proposition}\ \texttt{')'} \\
|\ & \texttt{'mcf'}\ \texttt{'('}\ \text{mcf}\ \texttt{')'}
\end{aligned}$$

$$\begin{aligned}
\text{response} ::= {}& [\texttt{'inevitably'}]\ \text{action\_formula} \\
& [\texttt{'before'}\ \text{action\_formula}] \\
& [\texttt{'unless'}\ \text{action\_formula}] \\
& [\texttt{'before'}\ \texttt{'*'}\ \text{boolean\_expression}] \\
& [\texttt{'unless'}\ \texttt{'*'}\ \text{boolean\_expression}] \\
\text{boolean\_expression} ::= {}& b[\text{data\_instance}/t] \\
\text{action\_formula} ::= {}& af[\texttt{'any'}/\,\textbf{true},\ \texttt{'paradox'}/\,\textbf{false}] \\
\text{mcf} ::= {}& \phi[\text{boolean\_expression}/b] \\
\text{regular\_af} ::= {}& \text{regular\_af}\ \texttt{'+'}\ \text{regular\_af} \\
|\ & \text{regular\_af}\ \texttt{'.'}\ \text{regular\_af} \\
|\ & af\ [\texttt{'*'}]
\end{aligned}$$

Here we use t, b, af, and ϕ from the modal mu-calculus syntax as defined in Appendix A.

5.3 Semantics of μ++

In creating this language, we have taken care to keep the concepts Monitor and Requirement rather separate. While these integrate nicely to create this language, they are essentially separate down to the lowest of levels. In fact, we decided to stage the translation to μ++ formulae by first translating the requirements, and then manipulating it a second time if it contains unbound (that is, monitor) variables.

5.3.1 Formalizing requirements

Assume we have an adjusted modal μ-calculus which automatically updates our variables for us every time the model takes an action (we will formalize this later in this chapter). This allows us to translate the requirements without having to worry about possible monitor variables. Note that if we do not use monitors, we can use the same definitions with the standard modal μ-calculus.

Definition 1 (Semantics of requirements)**.** Let ϕ be a requirement expression. We define the interpretation of requirement ϕ, notation $[\![\phi]\!]$ as an adapted modal μ-calculus formula, inductively by;

$$[\![\texttt{require: } \phi]\!] = \nu X(\texttt{init} : \mathsf{Bool} = \textbf{true}) \,.\, ([\textbf{true}]\, X(\textbf{false}) \wedge [\![\phi]\!]) \tag{5.1}$$

$$[\![\phi \;\; \phi]\!] = [\![\phi]\!] \wedge [\![\phi]\!] \tag{5.2}$$

$$[\![\texttt{if } b\texttt{: } \phi]\!] = b \rightarrow [\![\phi]\!] \tag{5.3}$$

$$[\![\texttt{for } d \texttt{ in } D\texttt{: } \phi]\!] = \forall d{:}D \,.\, ([\![\phi]\!]) \tag{5.4}$$

$$[\![\texttt{after } af\texttt{: } \phi]\!] = [af]\, [\![\phi]\!] \tag{5.5}$$

$$[\![\texttt{initially: } \phi]\!] = \texttt{init} \rightarrow [\![\phi]\!] \tag{5.6}$$

$$[\![\texttt{invariant: } \phi]\!] = [\![\phi]\!] \tag{5.7}$$

$$[\![\texttt{assert } \psi]\!] = [\![\psi]\!] \tag{5.8}$$

Definition 2 (Semantics of assertions)**.** Let ψ be an assertion body expression. We define the interpretation of assertion body ψ, notation $[\![\psi]\!]$ as an adapted modal μ-calculus formula, inductively by;

$$[\![\texttt{inevitably(}\psi\texttt{)}]\!] = \mu X \,.\, (([\textbf{true}]\, X \wedge \langle\textbf{true}\rangle\, \textbf{true}) \vee [\![\psi]\!]) \tag{5.9}$$

$$[\![\texttt{possible(}R\texttt{, } \psi\texttt{)}]\!] = \langle R\rangle\, [\![\psi]\!] \tag{5.10}$$

$$[\![\texttt{afterall(}R\texttt{, } \psi\texttt{)}]\!] = [R]\, [\![\psi]\!] \tag{5.11}$$

$$[\![b]\!] = b \tag{5.12}$$

$$[\![\texttt{mcf(}\varphi\texttt{)}]\!] = \varphi \tag{5.13}$$

$$\begin{aligned}
[\![\texttt{response}(\sigma)]\!] = \zeta(inev(\sigma))X\,.\,((& \\
&[\overline{af(\sigma)} \wedge \overline{ex(\sigma)}]\ X \wedge \\
&[er(\sigma) \wedge \overline{ex(\sigma)}]\ \textbf{false} \wedge \\
&\langle \textbf{true}^\star \cdot af(\sigma)\rangle\ \textbf{true} \wedge \\
&\overline{er{*}(\sigma)} \\
&) \vee ex{*}(\sigma))
\end{aligned} \tag{5.14}$$

$$\begin{aligned}
[\![\texttt{response}{*}(\sigma)]\!] = \zeta(inev(\sigma))X\,.\,((& \\
&[\overline{af(\sigma)} \wedge \overline{ex(\sigma)}]\ X \wedge \\
&[er(\sigma) \wedge \overline{ex(\sigma)}]\ \textbf{false} \wedge \\
&\overline{er{*}(\sigma)} \\
&) \vee ex{*}(\sigma))
\end{aligned} \tag{5.15}$$

Where we define the helper functions ζ, as well as $inev$, af, er, ex, $er*$, and $ex*$, as follows, such that ζ returns the appropriate type of fixed point, and the other functions return the encapsulated value of a response type if applicable, or **false** (or `''` for $inev$) if not;

$$\zeta(s) ::= \begin{cases} \mu & \text{if } s = \texttt{inevitably} \\ \nu & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\text{response} ::= &\left[\overset{inev}{\boxed{\texttt{'inevitably'}}}\right]\ \overset{af}{\boxed{\text{action\_formula}}} \\
&\left[\texttt{'before'}\ \overset{er}{\boxed{\text{action\_formula}}}\right] \\
&\left[\texttt{'unless'}\ \overset{ex}{\boxed{\text{action\_formula}}}\right] \\
&\left[\texttt{'before'}\ \texttt{'*'}\ \overset{er*}{\boxed{\text{boolean\_expression}}}\right] \\
&\left[\texttt{'unless'}\ \texttt{'*'}\ \overset{ex*}{\boxed{\text{boolean\_expression}}}\right]
\end{aligned}$$

So, for example;

$$inev(\texttt{inevitably finish before error}) = \texttt{inevitably} \tag{5.16}$$
$$af(\texttt{inevitably finish before error}) = \texttt{finish} \tag{5.17}$$
$$er(\texttt{inevitably finish before error}) = \texttt{error} \tag{5.18}$$
$$ex(\texttt{inevitably finish before error}) = \textbf{false} \tag{5.19}$$
$$er{*}(\texttt{inevitably finish before error}) = \textbf{false} \tag{5.20}$$
$$ex{*}(\texttt{inevitably finish before error}) = \textbf{false} \tag{5.21}$$

One keyword was missing in the definition given above; `sequentially`. This is because it is quite a complex keyword, and is more easily translated to a number of responses instead. If we define a sequence σ of individual response blocks $[\sigma_1, \sigma_2, \ldots, \sigma_n]$, then we can define `sequentially` as follows;

$$\texttt{sequentially}[\sigma] = \bigwedge_{0<i\leq n} \texttt{response}\begin{pmatrix} inev(\sigma_i) \;\; af(\sigma_i) \\ \texttt{before} \;\; er(\sigma_i) \vee \bigcup_{i<j\leq n} af(\sigma_j) \\ \texttt{unless} \;\; ex(\sigma_i) \\ \texttt{before}* \;\; er*(\sigma_i) \\ \texttt{unless}* \;\; ex*(\sigma_i) \end{pmatrix} \tag{5.22}$$

$$\texttt{sequentially}*[\sigma] = \bigwedge_{0<i\leq n} \texttt{response}*\begin{pmatrix} inev(\sigma_i) \;\; af(\sigma_i) \\ \texttt{before} \;\; er(\sigma_i) \vee \bigcup_{i<j\leq n} af(\sigma_j) \\ \texttt{unless} \;\; ex(\sigma_i) \\ \texttt{before}* \;\; er*(\sigma_i) \\ \texttt{unless}* \;\; ex*(\sigma_i) \end{pmatrix} \tag{5.23}$$

Or, more plainly; for each individual response block in the given sequence, we write an equivalent individual `response` (or `response`$*$) keyword, with the exception that the `before` action formula is padded with the action formulas searched for in each of the sequences following the one we are translating.

5.3.2 Normalizing monitors

Monitors can come in quite complex and intricate shapes. To aid in the understanding and simplicity of the translation, we will first normalize each monitor to a simpler form. Specifically, this form contains no `if` and `otherwise` keywords. Additionally, `for` keywords may not be nested, and may only have a single nested `on` keyword. As such, we end up with monitors where the body consists solely of a list of `on` and `for ... on` constructions.

The first step in this process is to make the implicit `otherwise: X()` explicit. If the monitor already has an unnested `otherwise`, this step may be skipped. If there is no unnested `otherwise`, we must add an unnested `otherwise: X()` such that all behavior is described explicitly.

Secondly, we must resolve these `otherwise` keywords to their equivalent `on` keyword. Since the `otherwise` keyword should catch all actions that are not explicitly caught by the expressions in the same nesting as the `otherwise`, we can replace the `otherwise` with `on` $\overline{af}$ where $\overline{af}$ is the complement of all actions that are explicitly caught.

From here on, we will use 'scope' to describe this notion of nestedness. A scope is characterized by a lineage—a single path of 'parent' keywords starting with the monitor itself, of which each is contained in the scope of its predecessor. The reverse of the 'parent' relationship is referred to as 'child'. A direct child is a keyword that is unnested in the scope of the direct parent. If we consider each keyword a node in a tree, then the monitor is the root node, a lineage is the path to the direct parent keyword, and the scope is the forest of subtrees formed by the children of the direct parent keyword. As such, we may speak of relative rootedness within a scope.

To find $\overline{af}$, we introduce the notion of an inaction set of a monitor (and by extension, of a scope). This inaction set represents the action formula that an `otherwise` keyword responds to. More formally, the inaction set of a monitor (or scope) is the complete set of actions that is not acted upon within that monitor (or scope), except by a possible `otherwise` in the highest (sub)scope. It is constructed as the intersection of the complements of sets that *are* acted upon.

Finding this inaction set is quite involved, as there may be a lot going on inside a monitor. We will introduce the logic of keywords one by one and see how the definition of the inaction set evolves.

In the simplest case, a monitor contains a single `on` keyword. In this case, the inaction set should be the complement of actions described by that single `on` keyword. If we have multiple `on` keywords, then we should take the complement of actions described by any of the `on` keywords. We can construct this complement by taking the conjunction of complements of individual `on` keywords' action formulas. Constructing this set in this way allows us to build the inaction set recursively.

We can generalize from `on` keywords to any siblings (direct children of the same parent), and say that the inaction set of any two siblings is the conjunction of their individual inaction sets. As such, each direct child in a scope is transformed into the set of actions that it does not catch, and the conjunction of these is the set of actions which no child catches.

Following the same recursive mindset, we can use the boolean expression of an `if` keyword to imply the inaction set of its children. If the boolean expression is false, then the `if` will catch no actions and its inaction set is therefore **true**. If the boolean expression is true, then the inaction set is the conjunction of the childrens' inaction sets.

Similarly, the `for` keyword can be transformed to the conjunction of inaction sets of its children for each valid instantiation of the variables declared in the `for` keyword. Combining what we have so far, we get the following definition:

Definition 3 (Inaction set of a monitor without otherwise)**.** Let ψ be a monitor expression. We define the inaction set of a monitor ψ, notation $\llbracket\psi\rrbracket$ as an action formula, inductively by;

$$\llbracket \texttt{monitor}\ \ X(D\,d \leftarrow t)\texttt{:}\ \ \phi\rrbracket = \llbracket\phi\rrbracket \tag{5.24}$$

$$\llbracket\phi\ \ \phi\rrbracket = \llbracket\phi\rrbracket \wedge \llbracket\phi\rrbracket \tag{5.25}$$

$$\llbracket \texttt{if}\ \ b\texttt{:}\ \ \phi\rrbracket = b \rightarrow \llbracket\phi\rrbracket \tag{5.26}$$

$$\llbracket \texttt{for}\ \ d\ \ \texttt{in}\ \ D\texttt{:}\ \ \phi\rrbracket = \forall d{:}D\,.\,(\llbracket\phi\rrbracket) \tag{5.27}$$

$$\llbracket \texttt{on}\ \ af\texttt{:}\ \ X(d \leftarrow t)\rrbracket = \overline{af} \tag{5.28}$$

However, we have yet to include the effects of an `otherwise` keyword. This is not trivial, because we cannot transform an otherwise keyword using only itself and its siblings. As such, the recursive nature of definition 3 is working against us here. Conversely, we can exploit the nature of the `otherwise` keyword to help us out a litte.

Recall that the inaction set of a scope is defined as the set of actions a relatively rooted (unnested) `otherwise` keyword is supposed to catch. As such, we may ignore a relatively rooted `otherwise` keyword when calculating the inaction set of that scope. As with the `if` keyword, we can ignore a keyword by considering its inaction set to be **true**. In contrast, we should observe that the inaction set of a nested scope containing a relatively rooted `otherwise` keyword is **false**, as the `otherwise` in that scope will catch all actions that its siblings do not catch.

Consider the following example, which contains multiple `otherwise` keywords:

```
monitor x(Bool y = false, Int z = 0):
  if y == false:
    on a:       x(y = true)
    otherwise: x(z = z + 1)

  if y == true:
    on a: x(y = false)
    otherwise: x(z = 0)

  otherwise: x(z = z - 1)
```

Here, we would want to see that the first otherwise is transformed to `on !a: x(z=z+1)`, the second otherwise is transformed to `on !a: x(z=0)`, and the third otherwise is transformed to `on (y==false->(!a&&!!a))&&(y==true->(!a&&!!a)): x(z=z-1)`, or something that is equivalent.

This example highlights the fact that in any scope where an `otherwise` is present, all actions are caught. As such, we can simplify the inaction set of such a scope to **false**. We can do this by, as mentioned before, considering the inaction set of a nested `otherwise` keyword as **false**, as the conjunction with siblings will resolve the entire scope's inaction set to **false**.

The distinction between a relatively rooted `otherwise` and other `otherwise` keywords should also be clear from the example; It should be clear that when we transformed the first two `otherwise` keywords in the example, we ignored the effects of the `otherwise` keywords in their respective scopes. However, we did not ignore them when we transformed the third `otherwise` keyword.

In the following definition, we will encode the relative rootedness of the current scope ("is this the first scope we have entered?") by defining two operators $\llbracket\psi\rrbracket$ and $\llbracket\psi\rrbracket'$, such that $\llbracket\psi\rrbracket$ defines behavior for relative root-level scopes (where `otherwise` should have no effect), and $\llbracket\psi\rrbracket'$ defines behavior for deeper scopes (where `otherwise` should have an effect).

Definition 4 (Inaction set of a monitor)**.** Let ψ be a monitor expression. We define the inaction set (set of actions to which a monitor does not explicitly respond) of a monitor ψ, notation $\llbracket\psi\rrbracket$ as a action formula, inductively by;

$$\llbracket \texttt{monitor}\ X(D\,d \leftarrow t)\texttt{:}\ \phi\rrbracket = \llbracket\phi\rrbracket \tag{5.29}$$
$$\llbracket\phi\ \ \phi\rrbracket' = \llbracket\phi\rrbracket' \wedge \llbracket\phi\rrbracket' \tag{5.30}$$
$$\llbracket\phi\ \ \phi\rrbracket = \llbracket\phi\rrbracket \wedge \llbracket\phi\rrbracket \tag{5.31}$$
$$\llbracket\texttt{if}\ b\texttt{:}\ \phi\rrbracket' = \qquad \llbracket\texttt{if}\ b\texttt{:}\ \phi\rrbracket = b \rightarrow \llbracket\phi\rrbracket' \tag{5.32}$$
$$\llbracket\texttt{for}\ d\ \texttt{in}\ D\texttt{:}\ \phi\rrbracket' = \qquad \llbracket\texttt{for}\ d\ \texttt{in}\ D\texttt{:}\ \phi\rrbracket = \forall d{:}D\,.\,(\llbracket\phi\rrbracket') \tag{5.33}$$
$$\llbracket\texttt{on}\ af\texttt{:}\ X(d \leftarrow t)\rrbracket' = \qquad \llbracket\texttt{on}\ af\texttt{:}\ X(d \leftarrow t)\rrbracket = \overline{af} \tag{5.34}$$
$$\llbracket\texttt{otherwise}\texttt{:}\ X(d \leftarrow t)\rrbracket' = \textbf{true} \tag{5.35}$$
$$\llbracket\texttt{otherwise}\texttt{:}\ X(d \leftarrow t)\rrbracket = \overline{\textbf{false}} \tag{5.36}$$

We can see that the definitions for the other keywords (5.29–5.34) stay mostly the same; we have added a prime version of the translation which means that we are in a 'deeper' scope than where we started. It should be noted that from a non-prime interpretation, we only go to a prime interpretation after an `if` (5.32) or `for` (5.33) keyword. With sibling nodes we keep a non-prime interpretation in a non-prime interpretation (5.31) and a prime interpretation in a prime interpretation (5.30).

Here we have simplified the role of the `otherwise` to resemble all actions in that scope, rather than just the inaction set of that scope. This does not actually matter for the semantics; regardless of the weaker definition, a 'deep' scope with an otherwise will have an empty inaction set.

We can now transform any `otherwise` keyword to an equivalent `on` keyword by calculating the inaction set of the scope formed by all siblings of that `otherwise` keyword, and using this result as the action formula of the `on` keyword.

To simplify the structure of the monitor to the shape we have detailed earlier on, we take four steps: distribute children such that each keyword has at most one direct child, hoist each `for` keyword to the highest scope, eliminate `if` keywords, and finally eliminate `for` keywords.

To distribute the children, we use the following transformations:

$$\texttt{if b: (}\phi_1\ \phi_2\texttt{)} \mapsto \texttt{(if b: }\phi_1\texttt{) (if b: }\phi_2\texttt{)} \tag{5.37}$$

$$\texttt{for d in D: (}\phi_1\ \phi_2\texttt{)} \mapsto \texttt{(for d in D: }\phi_1\texttt{) (for d in D: }\phi_2\texttt{)} \tag{5.38}$$

We can apply these transformations until each keyword (except for the monitor itself) has at most one child. It should be clear that this is a finite process with a unique result.

To hoist each `for` keyword to the highest scope, we must combine nested `for` keywords into one, and swap `if` and `for` keywords if `for` is the child of an `if` keyword. As long as we disallow the ghosting of variables (that is, introduce a new variable using a symbol which is already in use in this context) this transformation is safe.

$$\texttt{for d1' in D1: for d2' in D2: }\phi \mapsto \texttt{for d1' in D1, d2' in D2: }\phi \tag{5.39}$$

$$\texttt{if b: for d in D: }\phi \mapsto \texttt{for d in D: if b: }\phi \tag{5.40}$$

If we apply these transformations until no more are possible, the result should be a monitor which has only unnested `for` keywords. Again, this process is finite and produces a unique result.

We can eliminate `if` keywords by observing that we can encode their function into a nested `on` keyword. Since at this point all `if` keywords must have a (possibly indirect) child `on` keyword and no `for` children, we can use the following transformation to achieve the desired result. Again, this process is finite and produces a unique result.

$$\texttt{if } b\texttt{: on } af\texttt{: }\phi \mapsto \texttt{on } b \wedge af\texttt{: }\phi \tag{5.41}$$

Finally, we can eliminate some of the `for` keywords provided that the variables they introduce do not occur in the valuation change expression that follows the child `on` keyword.

$$\texttt{for d in D: on } af\texttt{: }\phi \mapsto \texttt{on } \exists\text{d:D}\,.\,(af)\texttt{: }\phi \qquad \text{if d} \notin \phi \tag{5.42}$$

A small example of all these transformations at work is illustrated here:

```
% original
monitor somemonitor(Int d = 0):
  for d' in Int:
    if d' != d:
      on a(d'): somemonitor(d=1)
      on b(d'): somemonitor(d=d')
```

```
% normalized
monitor somemonitor(Int d = 0):
  on exists d':Int . (d' != d) &&   a(d')          : somemonitor(d=1)
  for d' in Int:
    on               (d' != d)             &&  b(d') : somemonitor(d=d')
  on forall d':Int . (d' != d) => (!a(d')) && !b(d')): somemonitor()
```

We refer to this simplified monitor as ‘normalized’. From here on, we will reason exclusively with normalized monitors. The more complex forms are regarded as syntactical sugar and, as demonstrated by the normalization transformation, do not add any functionality.

5.3.3 Formalizing monitors

Monitors define the behavior through which the values of variables are updated upon following an action transition. As seen in the previous subsection, we can simplify the structure of these monitors to a simple, normalized form. In this normalized form, we have a series of descriptions of which actions allow certain valuation changes. These descriptions each consist of a possible quantification given through a `for` keyword, and must contain an action formula upon which the valuation change may be applied. This valuation change is described as a mapping of the monitor variables to an expression using any quantified variables and monitor variables.

These monitor variables are of the shape $i.j$, where i denotes the name of the monitor of which a variable is accessed, and j denotes the monitor-specific variable name. As such, we have a set of monitor names $\mathcal{X}$, and for each monitor $i \in \mathcal{X}$ we have a set of variable names $\mathcal{X}_i$. We call a variable a monitor variable if it is unbound and has the shape $i.j$, where $i \in \mathcal{X}$ and $j \in \mathcal{X}_i$. We group all defined variables in the set $\mathcal{V}$, with disjoint subsets $\mathcal{V}_n$ and $\mathcal{V}_m$ for normal variables and monitor variables respectively. In particular, we have $\not\psi \notin \mathcal{V}$.

These monitor variables keep track of interesting (data) expressions. A monitor is said to have a state, with the expressions $e \in \mathcal{E}$ that these monitor variables represent an encoding of that state. These expressions have a value. Therefore, we say that the valuation function σ_i is the state of a monitor i if for every $j \in \mathcal{X}_i$, $\sigma_i(j)$ results in the value of the expression e that the variable $i.j$ represents. We will use $\sigma = \sigma_{i_1} \bowtie \cdots \bowtie \sigma_{i_n}$ to represent the global state — that is, a single function that gives the matching expression for each variable, such that $\sigma(i.j) = \sigma_i(j)$ for all $i \in \mathcal{X}$ and $j \in \mathcal{X}_i$.

We formalize the syntax of a monitor $m_i \in M$ as a set Mon$_i$ of vectors $(\varkappa, af, f)$, with quantification $\varkappa$ (the `for` keyword), action formula af (the `on` keyword) and partial mapping $f : \mathcal{X}_i \to \mathcal{E}$ (updates in `i(...)`), in the following way

$$\text{Mon}_i \triangleq [\![\texttt{monitor i(}\cdots\texttt{):}\ \cdots]\!] \tag{5.43}$$

$$[\![\texttt{monitor i(}\cdots\texttt{):}\ \phi]\!] = [\![\phi]\!]^{\{\not\psi:\mathbf{0}\}} \tag{5.44}$$

$$[\![\phi_1\ \phi_2]\!]^{\varkappa} = [\![\phi_1]\!]^{\varkappa} \cup [\![\phi_2]\!]^{\varkappa} \tag{5.45}$$

$$[\![\texttt{for}\ d_1\ \texttt{in}\ D_1\texttt{,}\ \cdots\texttt{:}\ \phi]\!]^{\varkappa} = [\![\phi]\!]^{\varkappa \cup \{d_1 : D_1, \cdots\}} \tag{5.46}$$

$$[\![\texttt{on}\ af\texttt{:}\ x(v_1 = e_{v_1}, \cdots)]\!]^{\varkappa} = \left\{ (\varkappa, af, f) \,\middle|\, \forall_{j \in \mathcal{X}_i} \begin{pmatrix} f(j) = e_j & \text{if } \exists_k (j = v_k) \\ f(j) = j & \text{otherwise} \end{pmatrix} \right\} \tag{5.47}$$

For ease of use, we define the following shorthand notation

$$\text{Mon}_M \triangleq \left\{ \begin{pmatrix} \varkappa_1 \cup \cdots \cup \varkappa_n, \\ af_1 \wedge \cdots \wedge af_n, \\ f_1 \bowtie \cdots \bowtie f_n \end{pmatrix} \,\middle|\, \forall_{i \in \mathcal{X}}\, (\varkappa_i, af_i, f_i) \in \text{Mon}_i \right\} \tag{5.48}$$

where we again use the notation $(f_1 \bowtie \cdots \bowtie f_n)(i.j) = f_i(j)$.

5.3.4 Synchronizing monitors to a process LTS

We translate all monitors $m_i \in M$ to a single (combined) LTS $\mathbb{M}$ as 3-tuple $(\Sigma, \longrightarrow_M, \sigma^*)$, where Σ is the set of all possible valuations σ, $\longrightarrow_M \subseteq \Sigma \times Act \times \Sigma$ is a transition relation, with $\sigma \xrightarrow{\alpha}_M \sigma'$ shorthand for $(\sigma, \alpha, \sigma') \in \longrightarrow_M$, and $\sigma^* \in \Sigma$ is the initial state. We require that the transition relation provides a target for each valid action from each state. That is, $\forall_{\sigma \in \Sigma} \forall_{\alpha \in Act} \exists_{\sigma' \in \Sigma} (\sigma \xrightarrow{\alpha}_M \sigma')$.

From the monitor specifications we can derive partial state space and initial state trivially: from `monitor i(D1 d1 = t1, ..., Dn dn = tn): ...`, we get

$$\Sigma_i \triangleq D_1 \times \cdots \times D_n \tag{5.49}$$

$$\sigma_i^* \triangleq i.d_1 \mapsto [\![t_1]\!] \bowtie \cdots \bowtie i.d_n \mapsto [\![t_n]\!] \tag{5.50}$$

The LTS follows from the partial definitions and Mon_M

$$\Sigma \triangleq \Sigma_{m_1} \times \cdots \times \Sigma_{m_{|M|}} \tag{5.51}$$

$$\sigma^* \triangleq \sigma^*_{m_1} \bowtie \cdots \bowtie \sigma^*_{m_{|M|}} \tag{5.52}$$

$$\sigma \xrightarrow{\alpha}_M \sigma' \iff \exists_{(\varkappa, af, f) \in \text{Mon}_M} \left(\alpha \in [\![\exists \varkappa \,.\, af]\!]^\sigma \wedge \forall_{i \in [\sigma]} \forall_{j \in [\sigma_i]} \left([\![f(i.j)]\!]^\sigma = [\![i.j]\!]^{\sigma'} \right) \right) \tag{5.53}$$

Note that due to this definition, we automatically satisfy $\forall_{\sigma \in \Sigma} \forall_{\alpha \in Act} \exists_{\sigma' \in \Sigma} (\sigma \xrightarrow{\alpha}_M \sigma')$ as monitors have an implicit `otherwise` which requires that all actions have at least one corresponding item in Mon_M.

Assume we have a process given as LTS, $\mathbb{P} = (S, \longrightarrow_P, s^*)$. We can create a synchronized LTS $\mathbb{S} = (S \times \Sigma, \longrightarrow_S, (s^*, \sigma^*))$, such that

$$(s, \sigma) \xrightarrow{\alpha}_S (s', \sigma') \iff s \xrightarrow{\alpha}_P s' \wedge \sigma \xrightarrow{\alpha}_M \sigma' \tag{5.54}$$

5.3.5 Adapting the modal mu-calculus

We propose an extramural extension to the modal μ-calculus, which additionally considers the data encoded in the states of a monitor-synchronized LTS. In particular, the modal μ-calculus as described in Appendix A is a special case where we consider the trivial monitor $\mathbf{Mon_0} \triangleq \{(\{\not{v}{:}\mathbf{0}\}, \textbf{true}, \not{v} \mapsto 0)\}$ which defines no data nor valuation in any state (or, more specifically, a quantification and valuation over the unused variable $\not{v}$).

We interpret a extramural modal μ-calculus formula in the context of a synchronized LTS $\mathbb{S}$. We will subscript the name of the (synchronized) LTS to the interpretation to make the distinction between this and the regular modal μ-calculus. The interpretation of variable expressions (and action formulas) get no such subscript, as they do not depend on these contexts—and to visually distinguish them. The interpretation (semantics) of this extramural modal μ-calculus is defined as follows:

$$[\![b]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \begin{cases} S & \text{if } (\hat{\sigma} \cup \sigma)(b) = \textbf{true} \\ \emptyset & \text{if } (\hat{\sigma} \cup \sigma)(b) = \textbf{false} \end{cases} \tag{5.55}$$

$$[\![\top]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = S \tag{5.56}$$

$$[\![\bot]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \emptyset \tag{5.57}$$

$$[\![\neg\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = S \setminus [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \tag{5.58}$$

$$[\![\phi_1 \wedge \phi_2]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = [\![\phi_1]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \cap [\![\phi_2]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \tag{5.59}$$

$$[\![\phi_1 \vee \phi_2]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = [\![\phi_1]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \cup [\![\phi_2]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \tag{5.60}$$

$$[\![\langle af\rangle\, \phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \left\{ s \;\middle|\; \exists \begin{pmatrix} (s',\sigma') \in S \times \Sigma, \\ \alpha \in [\![af]\!]^{\hat{\sigma},\sigma} \end{pmatrix} . \begin{pmatrix} (s,\sigma) \xrightarrow{\alpha} (s',\sigma') \;\wedge \\ s' \in [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma',\rho} \end{pmatrix} \right\} \tag{5.61}$$

$$[\![[af]\, \phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \left\{ s \;\middle|\; \forall \begin{pmatrix} (s',\sigma') \in S \times \Sigma, \\ \alpha \in [\![af]\!]^{\hat{\sigma},\sigma} \end{pmatrix} . \begin{pmatrix} (s,\sigma) \xrightarrow{\alpha} (s',\sigma') \Rightarrow \\ s' \in [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma',\rho} \end{pmatrix} \right\} \tag{5.62}$$

$$[\![\exists\, x{:}D\ .\ \phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \bigcup_{d \in \mathcal{M}_D} [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma}[d/x],\sigma,\rho} \tag{5.63}$$

$$[\![\forall\, x{:}D\ .\ \phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \bigcap_{d \in \mathcal{M}_D} [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma}[d/x],\sigma,\rho} \tag{5.64}$$

$$[\![\nu X(x{:}D \leftarrow t)\ .\ \phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \bigcup_{f \in (\Sigma,\mathcal{M}_D) \to 2^S} \left\{ f(\sigma, [\![t]\!]^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d' \in \mathcal{M}_D \end{pmatrix} . \left(f(\sigma', d) = [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma}[d/x],\sigma',\rho[X \leftarrow f]} \right) \right\} \tag{5.65}$$

$$[\![\mu X(x{:}D \leftarrow t)\ .\ \phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \bigcap_{f \in (\Sigma,\mathcal{M}_D) \to 2^S} \left\{ f(\sigma, [\![t]\!]^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d' \in \mathcal{M}_D \end{pmatrix} . \left(f(\sigma', d) = [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma}[d/x],\sigma',\rho[X \leftarrow f]} \right) \right\} \tag{5.66}$$

$$[\![X(t)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = \rho(X)(\sigma, [\![t]\!]^{\hat{\sigma},\sigma}) \tag{5.67}$$

5.3.6 Translating to mCRL2

To include the monitor semantics into the modal μ-calculus formula, we transform an arbitrary extramural modal μ-calculus formula ϕ to a modal μ-calculus formula $tr(f, \phi)$, where $f : \mathcal{V}_m \to \mathcal{E}_n$ represents some substitution of monitor variables to expressions without monitor variables. We define the function $tr(f, \phi)$ recursively as follows. Without loss of generality, we can assume that our monitors' states can be represented using a single variable v_m of type D_m.

$$tr(f, b) = f(b) \tag{5.68}$$
$$tr(f, \top) = \top \tag{5.69}$$
$$tr(f, \bot) = \bot \tag{5.70}$$
$$tr(f, \neg\phi) = \neg tr(f, \phi) \tag{5.71}$$
$$tr(f, \phi \wedge \varphi) = tr(f, \phi) \wedge tr(f, \varphi) \tag{5.72}$$
$$tr(f, \phi \vee \varphi) = tr(f, \phi) \vee tr(f, \varphi) \tag{5.73}$$
$$tr(f, \langle af \rangle\, \phi) = \bigvee\nolimits_{(\varkappa, af', f') \in \mathsf{Mon}_M} (\exists\, \varkappa \,.\, \langle f(af \wedge af') \rangle\, tr(f \circ f', \phi)) \tag{5.74}$$
$$tr(f, [af]\, \phi) = \bigwedge\nolimits_{(\varkappa, af', f') \in \mathsf{Mon}_M} (\forall\, \varkappa \,.\, [f(af \wedge af')]\, tr(f \circ f', \phi)) \tag{5.75}$$
$$tr(f, \exists\, x{:}D \,.\, (\phi)) = \exists\, x{:}D \,.\, (tr(f, \phi)) \tag{5.76}$$
$$tr(f, \forall\, x{:}D \,.\, (\phi)) = \forall\, x{:}D \,.\, (tr(f, \phi)) \tag{5.77}$$
$$tr(f, \nu X(x{:}D \leftarrow t) \,.\, \phi) = \nu X(v'_m{:}D_m \leftarrow f(v_m), x{:}D \leftarrow f(t)) \,.\, tr(h, \phi) \tag{5.78}$$
$$tr(f, \mu X(x{:}D \leftarrow t) \,.\, \phi) = \mu X(v'_m{:}D_m \leftarrow f(v_m), x{:}D \leftarrow f(t)) \,.\, tr(h, \phi) \tag{5.79}$$
$$tr(f, X(t)) = X(f(v_m), f(t)) \tag{5.80}$$

where

$$(f \circ f')(x) \triangleq f(f'(x)) \tag{5.81}$$
$$h(v) \triangleq \begin{cases} v & \text{if } v \in \mathcal{V}_n \\ v' & \text{if } v \in \mathcal{V}_m \end{cases} \tag{5.82}$$

Note that the translation merely encodes the state of the monitor into its fixed points and updates using the modalities.

5.3.7 Proof of equivalence

Definition 5 (f_σ)**.** Let σ be a monitor variable valuation. We can infer from the definition of this σ (and σ from) a transformation function f_σ such that for any monitor variable $v_m \in \mathcal{V}_m$ we have $[\![v_m]\!]^\sigma = [\![f_\sigma(v_m)]\!]$.

Definition 6 (d_σ)**.** Let σ be a monitor variable valuation. We can infer from the definition of this σ (and σ from) a concrete data value d_σ such that for any monitor variable $v \in \mathcal{V}_m$ we have $\sigma(v) = \text{proj}_v(d_\sigma)$, where proj_v is some structured projection function (e.g. indexing on a tuple). Note that we can construct such a d_σ from f_σ and vice versa, since $[\![f_\sigma(v)]\!] = \sigma(v) = \text{proj}_v(d_\sigma)$ for $v \in \mathcal{V}_m$.

Definition 7 ($\rho\!\downarrow$)**.** Let ρ be a logical variable valuation. We can infer from the definition of this ρ (and ρ from) an extended valuation $\rho\!\downarrow$ such that for all X, σ, σ' we have

$$\rho\!\downarrow\!(X) = p\!\downarrow \iff \rho(X) = p \tag{5.83}$$

$$p\!\downarrow\!(\sigma', d_\sigma, d) \triangleq p(\sigma, d) \tag{5.84}$$

Lemma 1 (translation equivalence)**.** *Let $\mathbb{S}$ be a synchronized LTS of a process LTS $\mathbb{P}$ and a monitor LTS $\mathbb{M}$. For any extramural modal μ-calculus formula ϕ, valuation $\hat{\sigma}$, logical variable valuation ρ, and state σ in $\mathbb{M}$, it holds that*

$$[\![tr(f_\sigma, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} = [\![tr(f_\sigma, \phi)]\!]_{\mathbb{P}}^{\hat{\sigma},\rho} \tag{5.85}$$

Proof. Due to the definition of f_σ, we know that f_σ will translate all monitor variables to some expression not containing monitor variables, as we assume that the result of applying f_σ on some data expression t can be interpreted without a monitor context.

In the definition of $tr(f, \phi)$, we can see that all places where monitor variables can occur is processed through applying the substitution function f on it. This f can take one of two shapes: $f_\sigma[\circ f' \cdots]$, or $h[\circ f' \cdots]$. In either case, the resulting expression cannot contain monitor variables by definition. As such, we can conclude that the result of $tr(f_\sigma, \phi)$ does not contain monitor variables, and therefore its interpretation is equal in $\mathbb{S}$ and $\mathbb{P}$. ■

Lemma 2 (f_σ-elimination)**.** *Let $\mathbb{S}$ be a synchronized LTS of a process LTS $\mathbb{P}$ and a monitor LTS $\mathbb{M}$. For any data expression t, valuation $\hat{\sigma}$, and state σ in $\mathbb{M}$, it holds that*

$$(\hat{\sigma} \cup \sigma)(f_\sigma(t)) = (\hat{\sigma} \cup \sigma)(t) \tag{5.86}$$

Proof. The definition of f_σ tell us that the result of $f_\sigma(t)$ does not contain monitor variables. As such

$$\begin{aligned}(\hat{\sigma} \cup \sigma)(f_\sigma(t)) &= (\hat{\sigma})(f_\sigma(t)) \\ &= [\![f_\sigma(t)]\!]_{\mathbb{P}}^{\hat{\sigma},\rho} \\ &= [\![t]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \\ &= (\hat{\sigma} \cup \sigma)(t) \qquad ■\end{aligned} \tag{5.87}$$

Lemma 3 (f_σ action saturation). *Let $(s,\sigma),(s',\sigma')$ be states in $\mathbb{S}$, af some action formula, $(\varkappa', af', f')$ an element in Mon$_M$, and α some action in $[\![af \wedge af']\!]^{\hat{\sigma},\sigma}$ such that*

$$f_\sigma \circ f' \neq f_{\sigma'} \wedge (s,\sigma) \xrightarrow{\alpha} (s',\sigma') \tag{5.88}$$

Then, by virtue of $(s,\sigma) \xrightarrow{\alpha} (s',\sigma')$ there exists some $(\varkappa'', af'', f'') \in$ Mon$_M$ such that

$$f_\sigma \circ f'' = f_{\sigma'} \wedge a \in [\![af \wedge af'']\!]^{\hat{\sigma},\sigma} \tag{5.89}$$

And, by virtue of $(\varkappa', af', f') \in$ Mon$_M$ there exists state (s',σ'') such that

$$f_\sigma \circ f' = f_{\sigma''} \wedge (s,\sigma) \xrightarrow{\alpha} (s',\sigma'') \quad \blacksquare \tag{5.90}$$

Theorem 4 (synchronization–translation equivalence). *Let $\mathbb{S}$ be a synchronized LTS of a process LTS $\mathbb{P}$ and a monitor LTS $\mathbb{M}$. For any extramural modal μ-calculus formula ϕ, valuation $\hat{\sigma}$, logical variable valuation ρ, and state (s,σ) in $\mathbb{S}$, it holds that*

$$s \in [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \iff s \in [\![tr(f_\sigma,\phi)]\!]_{\mathbb{P}}^{\hat{\sigma},\rho\downarrow} \tag{5.91}$$

Proof. The proof follows trivially from the claims that

$$\begin{aligned} [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} &\overset{?}{=} [\![tr(f_\sigma,\phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &(5.92)\\ &\overset{1}{=} [\![tr(f_\sigma,\phi)]\!]_{\mathbb{P}}^{\hat{\sigma},\rho\downarrow} &(5.93)\end{aligned}$$

We can prove equality (5.92) by induction on the structure of ϕ as follows

$$\begin{aligned} [\![tr(f_\sigma,b)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![f_\sigma(b)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow}\\ &= (\hat{\sigma} \cup \sigma)(f_\sigma(b))\\ &\overset{2}{=} (\hat{\sigma} \cup \sigma)(b)\\ &= [\![b]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \quad \blacksquare &(5.94)\end{aligned}$$

$$\begin{aligned} [\![tr(f_\sigma,\top)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![\top]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow}\\ &= S\\ &= [\![\top]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \quad \blacksquare &(5.95)\end{aligned}$$

$$\begin{aligned} [\![tr(f_\sigma,\bot)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![\bot]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow}\\ &= \emptyset\\ &= [\![\bot]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \quad \blacksquare &(5.96)\end{aligned}$$

$$\begin{aligned} [\![tr(f_\sigma,\neg\phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![\neg tr(f_\sigma,\phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow}\\ &= S \setminus [\![tr(f_\sigma,\phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow}\\ &\overset{IH}{=} S \setminus [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho}\\ &= [\![\neg\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \quad \blacksquare &(5.97)\end{aligned}$$

$$\begin{aligned}
[\![tr(f_\sigma, \phi \vee \varphi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![tr(f_\sigma, \phi) \vee tr(f_\sigma, \varphi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \\
&= [\![tr(f_\sigma, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \cup [\![tr(f_\sigma, \varphi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \\
&\overset{\mathit{IH}}{=} [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \cup [\![\varphi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \\
&= [\![\phi \vee \varphi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \qquad \blacksquare \tag{5.98}
\end{aligned}$$

$$\begin{aligned}
[\![tr(f_\sigma, \phi \wedge \varphi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![tr(f_\sigma, \phi) \wedge tr(f_\sigma, \varphi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \\
&= [\![tr(f_\sigma, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \cap [\![tr(f_\sigma, \varphi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \\
&\overset{\mathit{IH}}{=} [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \cap [\![\varphi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \\
&= [\![\phi \wedge \varphi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \qquad \blacksquare \tag{5.99}
\end{aligned}$$

$$\begin{aligned}
[\![tr(f_\sigma, \forall\, x{:}D \,.\, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![\forall\, x{:}D \,.\, tr(f_\sigma, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \\
&= \bigcap_{d \in \mathcal{M}_D} [\![tr(f_\sigma, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma}[d/x],\sigma,\rho\downarrow} \\
&\overset{\mathit{IH}}{=} \bigcap_{d \in \mathcal{M}_D} [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma}[d/x],\sigma,\rho} \\
&= [\![\forall\, x{:}D \,.\, \phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \qquad \blacksquare \tag{5.100}
\end{aligned}$$

$$\begin{aligned}
[\![tr(f_\sigma, \exists\, x{:}D \,.\, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![\exists\, x{:}D \,.\, tr(f_\sigma, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \\
&= \bigcup_{d \in \mathcal{M}_D} [\![tr(f_\sigma, \phi)]\!]_{\mathbb{S}}^{\hat{\sigma}[d/x],\sigma,\rho\downarrow} \\
&\overset{\mathit{IH}}{=} \bigcup_{d \in \mathcal{M}_D} [\![\phi]\!]_{\mathbb{S}}^{\hat{\sigma}[d/x],\sigma,\rho} \\
&= [\![\exists\, x{:}D \,.\, \phi]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \qquad \blacksquare \tag{5.101}
\end{aligned}$$

$$\begin{aligned}
[\![tr(f_\sigma, [af]\ \phi)]\!]_{\mathbb{S}}^{\hat\sigma,\sigma,\rho\downarrow} &= \left[\!\!\left[\bigwedge\nolimits_{(\varkappa, af', f')\in\mathsf{Mon}_M}(\forall\, \varkappa\,.\,[f_\sigma(af \wedge af')]\ tr(f_\sigma \circ f', \phi))\right]\!\!\right]_{\mathbb{S}}^{\hat\sigma,\sigma,\rho\downarrow} \\
&= \bigcap_{(\varkappa, af', f')\in\mathsf{Mon}_M} [\![\forall\, \varkappa\,.\,[f_\sigma(af \wedge af')]\ tr(f_\sigma \circ f', \phi)]\!]_{\mathbb{S}}^{\hat\sigma,\sigma,\rho\downarrow} \\
&= \bigcap_{(\varkappa, af', f')\in\mathsf{Mon}_M,\, d\in\mathcal{M}_{D_\varkappa}} [\![[f_\sigma(af \wedge af')]\ tr(f_\sigma \circ f', \phi)]\!]_{\mathbb{S}}^{\hat\sigma[d/x_\varkappa],\sigma,\rho\downarrow} \\
&= \bigcap_{(\varkappa, af', f')\in\mathsf{Mon}_M,\, d\in\mathcal{M}_{D_\varkappa}} \left\{ s \;\middle|\; \forall\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![f_\sigma(af \wedge af')]\!]^{\hat\sigma[d/x_\varkappa],\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \Rightarrow \\ s' \in [\![tr(f_\sigma \circ f', \phi)]\!]^{\hat\sigma[d/x_\varkappa],\sigma',\rho\downarrow}\end{pmatrix} \right\} \\
&\overset{2}{=} \bigcap_{(\varkappa, af', f')\in\mathsf{Mon}_M,\, d\in\mathcal{M}_{D_\varkappa}} \left\{ s \;\middle|\; \forall\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![af \wedge af']\!]^{\hat\sigma[d/x_\varkappa],\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \Rightarrow \\ s' \in [\![tr(f_\sigma \circ f', \phi)]\!]^{\hat\sigma[d/x_\varkappa],\sigma',\rho\downarrow}\end{pmatrix} \right\} \\
&\overset{3}{=} \bigcap_{(\varkappa, af', f')\in\mathsf{Mon}_M,\, d\in\mathcal{M}_{D_\varkappa}} \left\{ s \;\middle|\; \forall\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![af \wedge af']\!]^{\hat\sigma[d/x_\varkappa],\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \Rightarrow \\ s' \in [\![tr(f_{\sigma'}, \phi)]\!]^{\hat\sigma,\sigma',\rho\downarrow}\end{pmatrix} \right\} \\
&= \left\{ s \;\middle|\; \forall\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![af]\!]^{\hat\sigma,\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \Rightarrow \\ s' \in [\![tr(f_{\sigma'}, \phi)]\!]^{\hat\sigma,\sigma',\rho\downarrow}\end{pmatrix} \right\} \\
&\overset{IH}{=} \left\{ s \;\middle|\; \forall\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![af]\!]^{\hat\sigma,\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \Rightarrow \\ s' \in [\![\phi]\!]^{\hat\sigma,\sigma',\rho}\end{pmatrix} \right\} \\
&= [\![[af]\ \phi]\!]_{\mathbb{S}}^{\hat\sigma,\sigma,\rho} \qquad \blacksquare \quad (5.102)
\end{aligned}$$

$$\begin{aligned}
[\![tr(f_\sigma, \langle af\rangle\ \phi)]\!]_{\mathbb{S}}^{\hat\sigma,\sigma,\rho\downarrow} &= \left[\!\!\left[\bigvee\nolimits_{(\varkappa, af', f')\in\mathsf{Mon}_M}(\exists\, \varkappa\,.\,\langle f_\sigma(af \wedge af')\rangle\ tr(f_\sigma \circ f', \phi))\right]\!\!\right]_{\mathbb{S}}^{\hat\sigma,\sigma,\rho\downarrow} \\
&= \bigcup_{(\varkappa, af', f')\in\mathsf{Mon}_M} [\![\exists\, \varkappa\,.\,\langle f_\sigma(af \wedge af')\rangle\ tr(f_\sigma \circ f', \phi)]\!]_{\mathbb{S}}^{\hat\sigma,\sigma,\rho\downarrow} \\
&= \bigcup_{(\varkappa, af', f')\in\mathsf{Mon}_M,\, d\in\mathcal{M}_{D_\varkappa}} [\![\langle f_\sigma(af \wedge af')\rangle\ tr(f_\sigma \circ f', \phi)]\!]_{\mathbb{S}}^{\hat\sigma[d/x_\varkappa],\sigma,\rho\downarrow} \\
&= \bigcup_{(\varkappa, af', f')\in\mathsf{Mon}_M,\, d\in\mathcal{M}_{D_\varkappa}} \left\{ s \;\middle|\; \exists\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![f_\sigma(af \wedge af')]\!]^{\hat\sigma[d/x_\varkappa],\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \wedge \\ s' \in [\![tr(f_\sigma \circ f', \phi)]\!]^{\hat\sigma[d/x_\varkappa],\sigma',\rho\downarrow}\end{pmatrix} \right\} \\
&\overset{2}{=} \bigcup_{(\varkappa, af', f')\in\mathsf{Mon}_M,\, d\in\mathcal{M}_{D_\varkappa}} \left\{ s \;\middle|\; \exists\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![af \wedge af']\!]^{\hat\sigma[d/x_\varkappa],\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \wedge \\ s' \in [\![tr(f_\sigma \circ f', \phi)]\!]^{\hat\sigma[d/x_\varkappa],\sigma',\rho\downarrow}\end{pmatrix} \right\} \\
&\overset{3}{=} \bigcup_{(\varkappa, af', f')\in\mathsf{Mon}_M,\, d\in\mathcal{M}_{D_\varkappa}} \left\{ s \;\middle|\; \exists\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![af \wedge af']\!]^{\hat\sigma[d/x_\varkappa],\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \wedge \\ s' \in [\![tr(f_{\sigma'}, \phi)]\!]^{\hat\sigma,\sigma',\rho\downarrow}\end{pmatrix} \right\} \\
&= \left\{ s \;\middle|\; \exists\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![af]\!]^{\hat\sigma,\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \wedge \\ s' \in [\![tr(f_{\sigma'}, \phi)]\!]^{\hat\sigma,\sigma',\rho\downarrow}\end{pmatrix} \right\} \\
&\overset{IH}{=} \left\{ s \;\middle|\; \exists\begin{pmatrix}(s', \sigma') \in S\times\Sigma, \\ \alpha \in [\![af]\!]^{\hat\sigma,\sigma}\end{pmatrix} . \begin{pmatrix}(s,\sigma) \xrightarrow{\alpha} (s',\sigma') \wedge \\ s' \in [\![\phi]\!]^{\hat\sigma,\sigma',\rho}\end{pmatrix} \right\} \\
&= [\![\langle af\rangle\ \phi]\!]_{\mathbb{S}}^{\hat\sigma,\sigma,\rho} \qquad \blacksquare \quad (5.103)
\end{aligned}$$

$$\begin{aligned}
[\![tr(f_\sigma, X(t))]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= [\![X(f_\sigma(v_m), f_\sigma(t))]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \\
&= \rho\downarrow(X)(\sigma, [\![f_\sigma(v_m)]\!]^{\hat{\sigma},\sigma}, [\![f_\sigma(t)]\!]^{\hat{\sigma},\sigma}) \\
&\overset{2}{=} \rho\downarrow(X)(\sigma, [\![v_m]\!]^{\hat{\sigma},\sigma}, [\![t]\!]^{\hat{\sigma},\sigma}) \\
&\overset{\rho\downarrow}{=} \rho(X)(\sigma, [\![t]\!]^{\hat{\sigma},\sigma}) \\
&= [\![X(t)]\!]_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \qquad \blacksquare \quad (5.104)
\end{aligned}$$

$$
\begin{aligned}
\llbracket tr(f_\sigma, \nu X(x{:}D \leftarrow t) \mathbin{.} \phi) \rrbracket_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} &= \llbracket \nu X(v'_m{:}D_m \leftarrow f_\sigma(v_m), x{:}D \leftarrow f_\sigma(t)) \mathbin{.} tr(h, \phi) \rrbracket_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho\downarrow} \\
&= \bigcup_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\downarrow(\sigma, \llbracket f_\sigma(v_m) \rrbracket^{\hat{\sigma},\sigma}, \llbracket f_\sigma(t) \rrbracket^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} . \left(p\downarrow(\sigma', d_{\sigma''}, d) = \llbracket tr(h, \phi) \rrbracket^{\hat{\sigma}[d_{\sigma''}/v'_m, d/x], \sigma', \rho\downarrow[X \leftarrow p\downarrow]} \right) \right\} \\
&\overset{2}{=} \bigcup_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\downarrow(\sigma, \llbracket v_m \rrbracket^{\hat{\sigma},\sigma}, \llbracket t \rrbracket^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} . \left(p\downarrow(\sigma', d_{\sigma''}, d) = \llbracket tr(h, \phi) \rrbracket^{\hat{\sigma}[d_{\sigma''}/v'_m, d/x], \sigma', \rho\downarrow[X \leftarrow p\downarrow]} \right) \right\} \\
&\overset{h}{=} \bigcup_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\downarrow(\sigma, \llbracket v_m \rrbracket^{\hat{\sigma},\sigma}, \llbracket t \rrbracket^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} . \left(p\downarrow(\sigma', d_{\sigma''}, d) = \llbracket tr(id, \phi) \rrbracket^{\hat{\sigma}[d/x], \sigma'[d_{\sigma''}/v_m], \rho\downarrow[X \leftarrow p\downarrow]} \right) \right\} \\
&\overset{d_\sigma}{=} \bigcup_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\downarrow(\sigma, \llbracket v_m \rrbracket^{\hat{\sigma},\sigma}, \llbracket t \rrbracket^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} . \left(p\downarrow(\sigma', d_{\sigma''}, d) = \llbracket tr(id, \phi) \rrbracket^{\hat{\sigma}[d/x], \sigma'', \rho\downarrow[X \leftarrow p\downarrow]} \right) \right\} \\
&\overset{2}{=} \bigcup_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\downarrow(\sigma, \llbracket v_m \rrbracket^{\hat{\sigma},\sigma}, \llbracket t \rrbracket^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} . \left(p\downarrow(\sigma', d_{\sigma''}, d) = \llbracket tr(f_{\sigma''}, \phi) \rrbracket^{\hat{\sigma}[d/x], \sigma'', \rho\downarrow[X \leftarrow p\downarrow]} \right) \right\} \\
&\overset{p\downarrow}{=} \bigcup_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p(\sigma, \llbracket t \rrbracket^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma'' \in \Sigma, \\ d \in \mathcal{M}_D \end{pmatrix} . \left(p(\sigma'', d) = \llbracket tr(f_{\sigma''}, \phi) \rrbracket^{\hat{\sigma}[d/x], \sigma'', \rho\downarrow[X \leftarrow p\downarrow]} \right) \right\} \\
&\overset{IH}{=} \bigcup_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p(\sigma, \llbracket t \rrbracket^{\hat{\sigma},\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma'' \in \Sigma, \\ d \in \mathcal{M}_D \end{pmatrix} . \left(p(\sigma'', d) = \llbracket \phi \rrbracket^{\hat{\sigma}[d/x], \sigma'', \rho[X \leftarrow p]} \right) \right\} \\
&= \llbracket \nu X(x{:}D \leftarrow t) \mathbin{.} \phi \rrbracket_{\mathbb{S}}^{\hat{\sigma},\sigma,\rho} \qquad \blacksquare \quad (5.105)
\end{aligned}
$$

$$
\begin{aligned}
\llbracket tr(f_\sigma, \mu X(x{:}D \leftarrow t)\ .\ \phi)\rrbracket_{\mathbb{S}}^{\hat\sigma,\sigma,\rho\downarrow} &= \llbracket \mu X(v'_m{:}D_m \leftarrow f_\sigma(v_m), x{:}D \leftarrow f_\sigma(t))\ .\ tr(h,\phi)\rrbracket_{\mathbb{S}}^{\hat\sigma,\sigma,\rho\downarrow} \\
&= \bigcap_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\!\downarrow\!(\sigma, \llbracket f_\sigma(v_m)\rrbracket^{\hat\sigma,\sigma}, \llbracket f_\sigma(t)\rrbracket^{\hat\sigma,\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} .\ \Big(p\!\downarrow\!(\sigma', d_{\sigma''}, d) = \llbracket tr(h,\phi)\rrbracket^{\hat\sigma[d_{\sigma''}/v'_m, d/x], \sigma', \rho\downarrow[X\leftarrow p\downarrow]} \Big) \right\} \\
&\overset{\underline{2}}{=} \bigcap_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\!\downarrow\!(\sigma, \llbracket v_m\rrbracket^{\hat\sigma,\sigma}, \llbracket t\rrbracket^{\hat\sigma,\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} .\ \Big(p\!\downarrow\!(\sigma', d_{\sigma''}, d) = \llbracket tr(h,\phi)\rrbracket^{\hat\sigma[d_{\sigma''}/v'_m, d/x], \sigma', \rho\downarrow[X\leftarrow p\downarrow]} \Big) \right\} \\
&\overset{\underline{h}}{=} \bigcap_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\!\downarrow\!(\sigma, \llbracket v_m\rrbracket^{\hat\sigma,\sigma}, \llbracket t\rrbracket^{\hat\sigma,\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} .\ \Big(p\!\downarrow\!(\sigma', d_{\sigma''}, d) = \llbracket tr(id,\phi)\rrbracket^{\hat\sigma[d/x], \sigma'[d_{\sigma''}/v_m], \rho\downarrow[X\leftarrow p\downarrow]} \Big) \right\} \\
&\overset{\underline{d_\sigma}}{=} \bigcap_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\!\downarrow\!(\sigma, \llbracket v_m\rrbracket^{\hat\sigma,\sigma}, \llbracket t\rrbracket^{\hat\sigma,\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} .\ \Big(p\!\downarrow\!(\sigma', d_{\sigma''}, d) = \llbracket tr(id,\phi)\rrbracket^{\hat\sigma[d/x], \sigma'', \rho\downarrow[X\leftarrow p\downarrow]} \Big) \right\} \\
&\overset{\underline{2}}{=} \bigcap_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p\!\downarrow\!(\sigma, \llbracket v_m\rrbracket^{\hat\sigma,\sigma}, \llbracket t\rrbracket^{\hat\sigma,\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma' \in \Sigma, \\ d_{\sigma''} \in \mathcal{M}_{D_m} \\ d \in \mathcal{M}_D \end{pmatrix} .\ \Big(p\!\downarrow\!(\sigma', d_{\sigma''}, d) = \llbracket tr(f_{\sigma''},\phi)\rrbracket^{\hat\sigma[d/x], \sigma'', \rho\downarrow[X\leftarrow p\downarrow]} \Big) \right\} \\
&\overset{\underline{p\downarrow}}{=} \bigcap_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p(\sigma, \llbracket t\rrbracket^{\hat\sigma,\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma'' \in \Sigma, \\ d \in \mathcal{M}_D \end{pmatrix} .\ \Big(p(\sigma'', d) = \llbracket tr(f_{\sigma''},\phi)\rrbracket^{\hat\sigma[d/x], \sigma'', \rho\downarrow[X\leftarrow p\downarrow]} \Big) \right\} \\
&\overset{\underline{IH}}{=} \bigcap_{p\in(\Sigma,\mathcal{M}_D)\to 2^S} \left\{ p(\sigma, \llbracket t\rrbracket^{\hat\sigma,\sigma}) \;\middle|\; \forall \begin{pmatrix} \sigma'' \in \Sigma, \\ d \in \mathcal{M}_D \end{pmatrix} .\ \Big(p(\sigma'', d) = \llbracket \phi\rrbracket^{\hat\sigma[d/x], \sigma'', \rho[X\leftarrow p]} \Big) \right\} \\
&= \llbracket \mu X(x{:}D \leftarrow t)\ .\ \phi\rrbracket_{\mathbb{S}}^{\hat\sigma,\sigma,\rho}
\end{aligned}
\qquad \blacksquare \ (5.106)
$$

5.3.8 A short example

Assume we have a simple train crossing system where we can detect emergencies and trains passing. We can model such a system as follows

```
act
  detected_emergency,
  train_passes_crossing;

proc
  normal =
    (train_passes_crossing . normal) +
    (detected_emergency . emergency) ;
  emergency =
    delta ;

init
  normal;
```

We might require of this system that a train may not pass the crossing when an emergency has been detected previously

```
monitor crossing(Bool emergency = false):
  on detected_emergency: crossing(emergency = true)

require:
  after train_passes_crossing:
    assert !crossing.emergency
```

Translating the requirement would give us

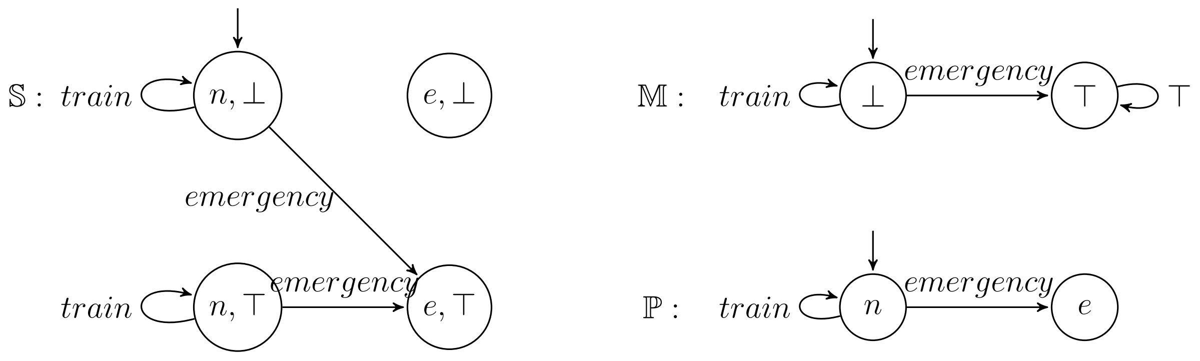
$$\begin{aligned}&\nu X(\text{init:Bool} \leftarrow \textbf{true}) \,.\, (\\ &\quad [\textbf{true}]\; X(\textbf{false}) \quad \wedge \\ &\quad [train]\; \neg\text{crossing.emergency}\\ &) \end{aligned} \tag{5.107}$$

Translating the monitor would give us

$$\sigma^* = \text{crossing.emergency} \mapsto \textbf{false} \tag{5.108}$$

$$\text{Mon}_M = \left\{ \begin{matrix} (\not{p}{:}\mathbf{0}, emergency, \text{crossing.emergency} \mapsto \textbf{true}), \\ (\not{p}{:}\mathbf{0}, \overline{emergency}, id) \end{matrix} \right\} \tag{5.109}$$

With LTSes $\mathbb{M}$, $\mathbb{P}$ and $\mathbb{S}$ as follows



The resulting requirement would be (where we shorten crossing.emergency to c.e for readability)

$$\begin{aligned}
& tr(\text{c.e} \mapsto \textbf{false}, \nu X(\text{init:Bool} \leftarrow \textbf{true}) \,.\, (\\
&\quad [\textbf{true}]\, X(\textbf{false}) \quad \wedge \\
&\quad [train]\, \neg\text{c.e} \quad)) && (5.110)\\
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow (\text{c.e} \mapsto \textbf{false})(\text{c.e}), \text{init:Bool} \leftarrow \textbf{true}) \,.\, tr(h, (\\
&\quad [\textbf{true}]\, X(\textbf{false}) \quad \wedge \\
&\quad [train]\, \neg\text{c.e} \quad)) && (5.111)\\
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \,.\, tr(h, (\\
&\quad [\textbf{true}]\, X(\textbf{false}) \quad \wedge \\
&\quad [train]\, \neg\text{c.e} \quad)) && (5.112)\\
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \,.\, (\\
&\quad tr(h, [\textbf{true}]\, X(\textbf{false})) \quad \wedge \\
&\quad tr(h, [train]\, \neg\text{c.e}) \quad) && (5.113)\\
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \,.\, (\\
&\quad [(\text{c.e} \mapsto \textbf{true})(emergency \wedge \textbf{true})]\, tr(h \circ (\text{c.e} \mapsto \textbf{true}), X(\textbf{false})) \quad \wedge \\
&\quad [id(\overline{emergency} \wedge \textbf{true})]\, tr(h \circ id, X(\textbf{false})) \quad \wedge \\
&\quad [(\text{c.e} \mapsto \textbf{true})(emergency \wedge train)]\, tr(h \circ (\text{c.e} \mapsto \textbf{true}), \neg\text{c.e}) \\
&\quad [id(\overline{emergency} \wedge train)]\, tr(h \circ id, \neg\text{c.e}) \quad) && (5.114)\\
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \,.\, (\\
&\quad [emergency \wedge \textbf{true}]\, tr(h \circ (\text{c.e} \mapsto \textbf{true}), X(\textbf{false})) \quad \wedge \\
&\quad [\overline{emergency} \wedge \textbf{true}]\, tr(h, X(\textbf{false})) \quad \wedge \\
&\quad [emergency \wedge train]\, tr(h \circ (\text{c.e} \mapsto \textbf{true}), \neg\text{c.e}) \\
&\quad [\overline{emergency} \wedge train]\, tr(h, \neg\text{c.e}) \quad) && (5.115)
\end{aligned}$$

$$\begin{aligned}
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \;.\; (\\
&\qquad [emergency]\; tr(h \circ (\text{c.e} \mapsto \textbf{true}), X(\textbf{false})) \quad \wedge \\
&\qquad [\overline{emergency}]\; tr(h, X(\textbf{false})) \quad \wedge \\
&\qquad [train]\; tr(h, \neg\text{c.e}) \quad)
\end{aligned} \tag{5.116}$$

$$\begin{aligned}
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \;.\; (\\
&\qquad [emergency]\; X((h \circ (\text{c.e} \mapsto \textbf{true}))(\text{c.e}), (h \circ (\text{c.e} \mapsto \textbf{true}))(\textbf{false})) \quad \wedge \\
&\qquad [\overline{emergency}]\; X(h(\text{c.e}), h(\textbf{false})) \quad \wedge \\
&\qquad [train]\; \neg tr(h, \text{c.e}) \quad)
\end{aligned} \tag{5.117}$$

$$\begin{aligned}
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \;.\; (\\
&\qquad [emergency]\; X(h((\text{c.e} \mapsto \textbf{true})(\text{c.e})), h((\text{c.e} \mapsto \textbf{true})(\textbf{false}))) \quad \wedge \\
&\qquad [\overline{emergency}]\; X(\text{c.e}', \textbf{false}) \quad \wedge \\
&\qquad [train]\; \neg h(\text{c.e}) \quad)
\end{aligned} \tag{5.118}$$

$$\begin{aligned}
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \;.\; (\\
&\qquad [emergency]\; X(h(\textbf{true}), h(\textbf{false})) \quad \wedge \\
&\qquad [\overline{emergency}]\; X(\text{c.e}', \textbf{false}) \quad \wedge \\
&\qquad [train]\; \neg\text{c.e}' \quad)
\end{aligned} \tag{5.119}$$

$$\begin{aligned}
&= \nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \;.\; (\\
&\qquad [emergency]\; X(\textbf{true}, \textbf{false}) \quad \wedge \\
&\qquad [\overline{emergency}]\; X(\text{c.e}', \textbf{false}) \quad \wedge \\
&\qquad [train]\; \neg\text{c.e}' \quad)
\end{aligned} \tag{5.120}$$

we can check whether $\llbracket \phi \rrbracket_{\mathbb{S}}^{\hat{\sigma},\sigma^*,\rho} = \llbracket tr(f_{\sigma^*}, \phi) \rrbracket_{\mathbb{P}}^{\hat{\sigma},\rho\downarrow}$ holds for this example. That is,

$$\left\llbracket \begin{aligned} &\nu X(\text{init:Bool} \leftarrow \textbf{true}) \;.\; (\\ &\quad [\textbf{true}]\; X(\textbf{false}) \quad \wedge \\ &\quad [train]\; \neg\text{c.e} \quad) \end{aligned} \right\rrbracket_{\mathbb{S}}^{\hat{\sigma},\sigma^*,\rho} \overset{?}{=} \left\llbracket \begin{aligned} &\nu X(\text{c.e}'\text{:Bool} \leftarrow \textbf{false}, \text{init:Bool} \leftarrow \textbf{true}) \;.\; (\\ &\quad [emergency]\; X(\textbf{true}, \textbf{false}) \quad \wedge \\ &\quad [\overline{emergency}]\; X(\text{c.e}', \textbf{false}) \quad \wedge \\ &\quad [train]\; \neg\text{c.e}' \quad) \end{aligned} \right\rrbracket_{\mathbb{P}}^{\hat{\sigma},\rho\downarrow} \tag{5.121}$$

$$= \{n, e\} \tag{5.122}$$

And indeed, when we solve for the fixed point using $\{n, e\}$, also the maximal set of $\mathbb{P}$, we see

$$\{n, e\} = \llbracket [\textbf{true}]\; X(\textbf{false}) \wedge [train]\; \neg\text{c.e} \rrbracket_{\mathbb{S}}^{\hat{\sigma}[\textbf{true}/\text{init}],\text{c.e}\mapsto\textbf{false},\rho[X\leftarrow\{n,e\}]} \tag{5.123}$$

$$= \{n, e\} \cap \llbracket [train]\; \neg\text{c.e} \rrbracket_{\mathbb{S}}^{\hat{\sigma},\text{c.e}\mapsto\textbf{false},\rho} \tag{5.124}$$

$$= \{n, e\} \cap \{n, e\} \tag{5.125}$$

$$= \{n, e\} \qquad \blacksquare \tag{5.126}$$

Chapter 6

Evaluation

6.1 Methodology

In this user experience evaluation, we observed four participants using our language and provided them with a short questionnaire before and afterwards. The participants were selected to have at least a Masters degree in Computer Science or Software Science. Two of the participants are engineers at ASML, the other two participants are PhD students at the Eindhoven University of Technology. At the start of each evaluation, we asked participants to rate their prior familiarity to a couple expertises on a four-point Likert scale. Figure 6.1 shows the obtained distribution of familiarities of the participants.

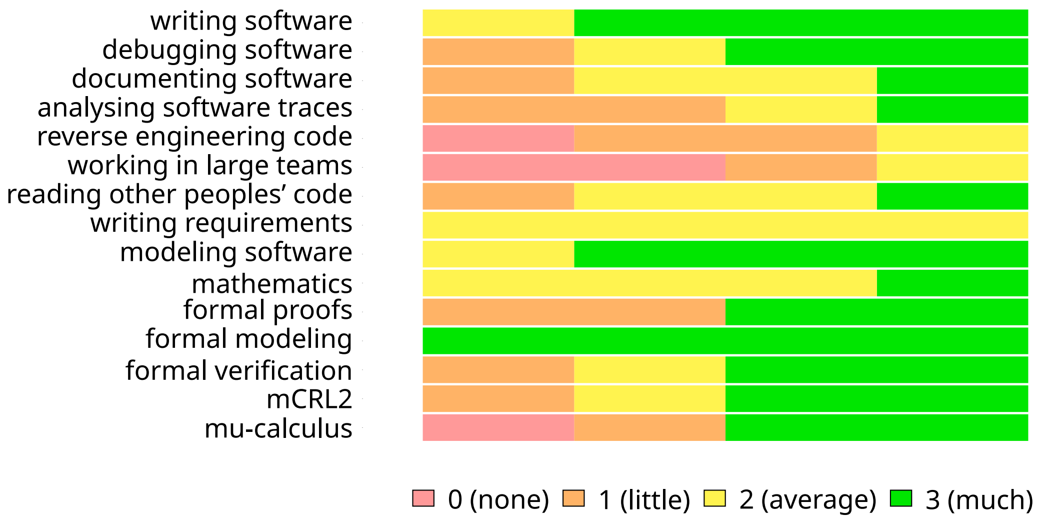


Figure 6.1: Participant expertise familiarity distribution

After the initial questionnaire, we provided the participants with a short (12 minute) presentation in which the features of the language were highlighted and explained briefly. Next, they were presented with a laptop with all the necessary tools preinstalled and configured. This configuration included Atom (a code editor), a μ++ parser and syntax highlighter, various quality-of-life tool support specifically for μ++, and 10 written natural language requirements.

Every requirement presented a property that the system must adhere to, a short reasoning for the requirements' existence, and suggestions of relevant action names.

During the observation, we monitored the participants use of the syntax. While we purposefully did not comment on the correctness of their use of syntax, we did answer any questions related to the meaning, use, or existence of certain features. This decision is motivated by the necessarily short introduction to our language: we cannot expect the participants to be immediately familiar with our language, we do not have the time for them to get familiar with our language, and this sort of information would usually be provided by a good documentation/manual.

Finally, we asked the participants to rate their experience with the language using the User Experience Questionnaire [19] and give a few comments as to their impression of the language. In this questionnaire, user experience is rated via 26 items on a seven-stage scale, each represented by two terms with opposite meanings. These items are weighted and combined to rate the user experience on six scales:

Attractiveness	Overall impression of the language. Do users like or dislike it?
Perspicuity	Is it easy to get familiar with the language and to learn how to use it?
Efficiency	Can users solve their tasks without unnecessary effort?
Dependability	Does the user feel in control of the interaction? Is it predictable?
Stimulation	Is it exciting and motivating to use the language? Is it fun to use?
Novelty	Is the design of the language creative? Does it catch the interest of users?

To evaluate the overall user experience, we use the benchmark from [20], comparing our tool on the six scales mentioned above against a dataset of 18483 participants across 401 studies concerning different products (business software, web pages, web shops, social networks). While these products are not directly comparable, the overall performance across these scales should give an indication of whether the language's user experience is comparable to common user experience ratings. This metric gives us a better picture on the quality of our tool and offers a first indicator for what is perceived as good and what types of improvements can be made.

6.2 Evaluation, observations and feedback

Figure 6.2 shows the User Experience Questionnaire results for the language user study. The comparison of the results for the tool with the data in the benchmark allows us to derive conclusions about the relative quality of the language compared to other tools [20].

Figure 6.2 shows along each of the axes the following data: the benchmark results are divided into five groups and encoded in the colored background of each axis as well as the Tukey box-and-whisker plot, and the evaluations given by the participants is displayed as a mean with 95% confidence interval. The evaluations are split into two groups, with a third group displaying the true mean. This is done because there is a big distinction between the answers given by those experienced with the modal mu-calculus, and those unexperienced with the modal mu-calculus.

Overall, we got very positive feedback from our participants. The UEQ results mostly reflect this. A clear outlier is the participant who professed a weak familiarity with modal mu-calculus, who mentioned that the low grades are most likely due to a severe lack of experience and the very high pace with which the presentation was done.

The rate of writing requirements was proportional to the professed level of familiarity with the modal μ-calculus. The participant with no modal μ-calculus experience was able to write three requirements in the allotted 1 hour and 40 minutes. The other three participants were able to write at least 10 requirements (one wrote some requirements in multiple ways).

Each of the following listings contains, as comments, the description that was given for that requirement to the participant, and additionally includes the unique (type of) solutions per requirement.

Note, the feedback of this evaluation has already been considered in writing this document. The `after` keyword used to be named `on` during testing, but this was deemed to be confusing due to the slight difference in behavior of the `on` keyword in a `monitor`. Finally, monitors and requirements used to specify exactly which monitors they would use internally with a `using` keyword, but this would amount to needless bookkeeping by the user as this information could trivially be found using a parse tree as well.

```
—— Alternating actions
  :: A railroad crossing gate is set up with a very simple design;
  :: It functions through telling the driving motor to rotate clockwise or
  :: counterclockwise by 90 degrees. It is therefore imperative that the
  :: openings and closings of said gate are strictly alternating.
  :: Assume the gate starts in the closed position.

  % actions:
  ~~ opening_gate
  ~~ closing_gate

% unique solution 1
monitor gate_position((struct closed | open) position = closed):
  on opening_gate: gate_position(position = open)
```

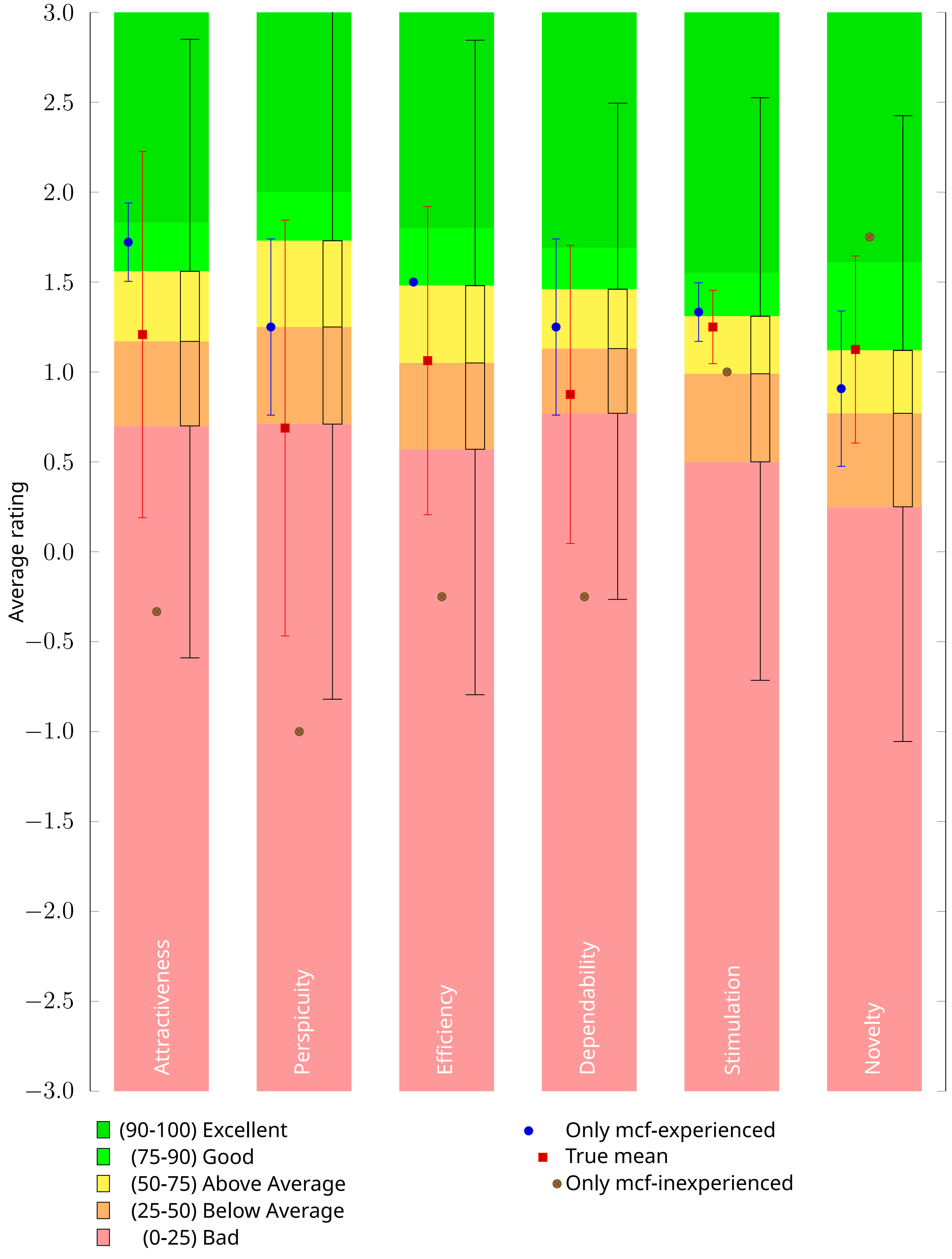


Figure 6.2: Experience benchmark

```
  on closing_gate: gate_position(position = closed)

require using gate_position:
  if gate_position.position == open:
    on opening_gate:
      assert false
  if gate_position.position == closed:
    on closing_gate:
      assert false

% unique solution 2
require:
  on opening_gate:
    assert response*(closing_gate before opening_gate)

  on closing_gate:
    assert response*(opening_gate before closing_gate)
```

```
-- Advance warning
  :: It is important that a railroad crossing's gate only starts closing after
  :: advance warning has been given to motorists and other traffic that want
  :: to pass the crossing.

  % actions:
  ~~ request_close_gate
  ~~ enable_warning_lights
  ~~ train_reserves_crossing

% unique solution 1
require:
  on train_reserves_crossing:
    assert response(inevitably request_close_gate)
    assert response(enable_warning_lights before request_close_gate)

% unique solution 2
require:
  on train_reserves_crossing:
    assert sequentially [
      enable_warning_lights,
      request_close_gate
    ]
```

The unique solutions presented above are not equivalent. However, because the requirement technically does not require that either action must happen (only that they must happen in order), both specifications are at least as strong as the requirement.

```
-- Immediate response to warning lights
  :: Enabling the warning lights must be immediately followed by a
  :: request to close the gate, such that the inconvenienced motorists
  :: are not conditioned to attempt a quick dash across the intersection
  :: when the warning lights turn on.

  % actions:
  ~~ request_close_gate
  ~~ enable_warning_lights

% unique solution 1
require:
  on enable_warning_lights:
    assert response(request_close_gate before !request_close_gate)
```

```
-- Ensure safe conditions on gate closing
  :: When the railroad crossing gate fails to close, for example when it is
  :: obstructed by a motorist illegally trying to evade the queue, the system
  :: must first ensure that the crossing is free of obstructions and the gate
  :: is back in the safe (fully open) position before reattempting closing the
  :: gate.

  % actions:
  ~~ request_close_gate
  ~~ closing_gate_failed
  ~~ crossing_is_empty
  ~~ gate_is_fully_open

% unique solution 1
require:
  on closing_gate_failed:
    assert response(crossing_is_empty before request_close_gate)
    assert response(gate_is_fully_open before request_close_gate)
    assert response(inevitably request_close_gate)
```

```
-- Open request before action
  :: It is important that the railroad crossing gates only open when a request
  :: for them to do so has been issued. (The other way around is inconvenient
  :: but not unsafe)

  % actions:
  ~~ request_open_gate
  ~~ opening_gate

% unique solution 1
monitor x(Bool y <- false):
  on request_open_gate:
    x(y=true)

  on opening_gate:
    x(y=false)

```

```
require using x:
  if !x.y:
    on opening_gate:
      assert false

% unique solution 2
monitor request_state(Bool requested = false, Bool prev_requested = false):
  on request_open_gate: request_state(requested = true, prev_requested = false)
  on opening_gate: request_state(requested = false, prev_requested = requested)
  otherwise: request_state(prev_requested = false)

require using request_state:
  on opening_gate:
    assert request_state.prev_requested == true

% unique solution 3
require:
  initially:
    assert response*(request_open_gate before opening_gate)

  on opening_gate:
    assert response*(request_open_gate before opening_gate)
```

```
-- Passage of trains is safe
  :: A train may only pass the intersection before the gates are opened or the
  :: warning lights are disabled.
  :: Assume that the gates are closed appropriately/safely.

  % actions:
  ~~ request_open_gate
  ~~ gate_is_fully_closed
  ~~ train_passes_crossing
  ~~ disable_warning_lights

% unique solution 1
require:
  on disable_warning_lights:
    assert response*(gate_is_fully_closed before train_passes_crossing)
  on request_open_gate:
    assert response*(gate_is_fully_closed before train_passes_crossing)

% unique solution 2
require:
  initially:
    assert response*(train_passes_crossing before request_open_gate)
    assert response*(train_passes_crossing before disable_warning_lights)
  on gate_is_fully_closed:
    assert response*(train_passes_crossing before request_open_gate)
    assert response*(train_passes_crossing before disable_warning_lights)
```

```
-- Trains only pass closed gates
  :: Trains may only pass the intersection when the gates are closed.
  :: Assume the gate starts in the open position.

  % actions:
  ~~ opening_gate
  ~~ closing_gate
  ~~ request_open_gate
  ~~ request_close_gate
  ~~ gate_is_fully_open
  ~~ gate_is_fully_closed
  ~~ train_passes_crossing

% unique solution 1
require:
  on opening_gate:
    assert response*(gate_is_fully_closed before train_passes_crossing)
  initially:
    assert response*(gate_is_fully_closed before train_passes_crossing)

% unique solution 2
monitor x(Int y <- 1):
  on gate_is_fully_closed:
    x(y=2)

  on opening_gate:
    x(y=1)

require using x:
  on train_passes_crossing:
    assert x.y==2
```

```
-- Minimal inconvenience
  :: The railroad crossing gates and warning lights should only attempt to
  :: close and disable respectively after a train requests safe passage
  :: through the crossing.

  % actions:
  ~~ request_close_gate
  ~~ train_passes_crossing
  ~~ disable_warning_lights
  ~~ train_reserves_crossing

% unique solution 1
require:
  on request_close_gate:
    assert response*(train_reserves_crossing before request_close_gate)
  on enable_warning_lights:
    assert response*(train_reserves_crossing before enable_warning_lights)
  initially:
    assert response*(train_reserves_crossing before request_close_gate)
    assert response*(train_reserves_crossing before enable_warning_lights)
  on train_reserves_crossing:
    assert response(inevitably request_close_gate)
    assert response(inevitably enable_warning_lights)

% unique solution 2
require:
  initially:
    assert response*(
      train_reserves_crossing
      before request_close_gate || disable_warning_lights
    )

  on train_passes_crossing:
    assert response*(
      train_reserves_crossing
      before request_close_gate || disable_warning_lights
    )
```

```
-- Eventually satisfies request
  :: Whenever a request to close the railroad crossing gates is made, it is
  :: eventually responded to by actually closing the gates.

  % actions:
  ~~ request_close_gate
  ~~ closing_gate

% unique solution 1
require:
  on request_close_gate:
    assert response(inevitably closing_gate)
```

```
-- Emergency halt
  :: An emergency can be called, in which case it is imperative that no more
  :: train crosses the intersection. The system should be completely reset by
  :: an on-location engineer (that is, the model may deadlock in pursuit of
  :: safety).

  % actions:
  ~~ detected_emergency
  ~~ train_passes_crossing

% unique solution 1
monitor emergency(Bool b = false):
  on detected_emergency: emergency(b = true)

require using emergency:
  on train_passes_crossing:
    assert !emergency.b
```

Chapter 7

Discussion

7.1 Implications and limitations

From the user experience evaluation, we see that for the participants with any familiarity with the modal μ-calculus, the μ++ language is quite a pleasant experience. The participant with no familiarity with the modal μ-calculus was less impressed, but admitted that this was mainly due to not being familiar with the type of reasoning that underpins the language as well as the very short introductory period. However, he was rather intrigued and thought that with a little more time he could learn the language quite well.

This seems to be the most positive result we could have hoped for. The language seems to give engineers who are familiar with similar methods a good overlap with the mental models they have built up. It does not scare away those who are new to the concept of formalizing requirements with its complexity.

Additionally, the speed with which the participants could write the requirements became noticeably faster as time progressed and they became more familiar with the syntax. However, it was already fast for those with modal μ-calculus familiarity to begin with. The earliest that any participant had written their first requirement was approximately 9 minutes after the introduction to the language was finished.

Most notably, each of the participants who is familiar with the modal μ-calculus has noted that it is a lot easier to work with this language than it is with the modal μ-calculus. This includes comments that μ++ is easier to read, more straightforward to find the relevant operators, and simpler to keep track of what is going on. This is quite impressive, considering that for all except the last requirement a unique solution was created that did not use a `monitor`. Recall that if we do not use a monitor, μ++ can be translated to modal μ-calculus quite straightforwardly.

7.2 Threats to validity

The pre-study in section 4 aimed to provide a solid overview of the kinds of requirements that are being written in the field, but the only sources are from students that are relatively new to writing requirements, and completely new to automated verification of systems. As such, we can make no claims that the requirements in the source material are at all reflective of those being written by verification experts, or even engineers with any verification experience. Additionally, the students are intrinsically rewarded by providing simpler and easier-to-check requirements, as the assignment provides plenty of opportunity for gaming the difficulty.

Where possible, the professor responsible for the course has guided the students through making concise and clear requirements, but from the amount of feedback given on (a small subset of) the source material it is difficult to imagine that these works represent that of the industry as a whole; it seems to indicate that the two groups make distinctly different kinds of mistakes, and use different language and concepts when attempting to describe the same system. As such, we have no basis upon which to account for mistakes due solely or in part to the language with which the to-be-translated requirements are written.

We claim that the resulting language, μ++, is easier to use than the modal μ-calculus, but this is only based on the remarks of people involved in the project, those who have attended presentations, and the four participants in the user experience evaluation who actually got to work with the language. As such, the sample size is fairly small. Additionally, most of these endorsements come from people who have an hour or two of experience with the language at best. While this is a promising result, none of these people will have had the time to encounter any issues or complexity in the language, nor will they know about the peculiarities and what it is like to really work with this language. As such, the potential is great and further research is very interesting.

Chapter 8

Conclusions

8.1 Master thesis objectives

To determine whether there is a need for a more approachable language to facilitate specifying and verifying formal requirements, we found alternatives in the modal μ-calculus, PSP, and Remenska. PSP and Remenska were not perfectly suited because they could not be applied broadly enough: many requirements could not be translated. To evaluate the usefulness of the modal μ-calculus, we used students' assignments in which they have to translate their own natural language requirements to the modal μ-calculus, and found that the students mistranslate around 42% of their requirements.

The majority of mistakes that were made are due to issues that the course material teaches how to overcome. Specifically, how to handle the case of *inevitability* and how this is different from, say, *always possible*. The cause is likely more complicated than just a poor understanding of the material. Indeed, even within the same report students may alternatingly make and not make similar mistakes for similar translations. This would suggest that, even though students may know how to address such issues when consciously aware of them, they lack a level of familiarity necessary to see these issues occurring naturally. That is; students do not make the mistake because they do not know how to do it better, but rather because they cannot spot the mistake in the first place. As such, we would expect that if we can somehow make the formulae more explicit and verbose in their notation—drawing attention to precisely such properties as *eventuality*, or the absence thereof, that we might see that students (and users of the μ-calculus formulae in general) are more easily aware of their own mistakes.

Therefore, these results seem to illustrate the need for a 'better' language; μ-calculus formulae are not friendly enough for practical use, and a more accessible language—one that easier to read and interpret—is necessary in order to make formal system verification via mCRL2 more approachable.

To guide the user of our new language μ++, we have attempted to create a syntax that is more verbose, which illustrates the intent of the requirement more transparently. This should help guide the user into writing better requirements as well. Furthermore, we have constrained the syntax of the modal μ-calculus so that it is a lot harder to write meaningless (that is, trivially true or false) syntax.

To prove that μ++ is indeed more accessible, we invited four participants with varying levels of familiarity with the modal μ-calculus to evaluate the experience of working with μ++. In two hours, each of the participants was able to learn the language and write at least three requirements. With active help to figure out the exact meaning of the syntax (which would usually come from a documentation), the written requirements were correctly translated. The participants with any familiarity with the μ-calculus admitted that μ++ is easier to work with.

8.2 Future Directions

A couple of things are still missing before μ++ can be used as a property specification language. First of all, the documentation that is provided in this document is not geared towards the target audience of the language itself. A more tailored user manual is needed for people who do not care about the implementation of the language but do want to work with it. We are of the opinion that μ++ is sufficiently user friendly such that users can indeed use the language without knowing the exact semantics.

Secondly, a more extensive user experience evaluation must be done. We could use the additional data points to consider where we can improve the syntax or add syntactic sugar. Additionally, we could more definitively determine whether μ++ is a good candidate to replace (or rather; interface with) the modal μ-calculus.

Thirdly, while we created a lexer and parser as part of the syntax highlighter used during the user experience evaluations, and the translation algorithm is described in this document, there does not currently exist a tool which actually performs the translation from μ++ to the modal μ-calculus.

Finally, it might be worth seeing whether μ++ can interface with modeling tools that translate their internal models to mCRL2. Most noticeably, these tools will likely use some sort of renaming scheme to translate their behavior into mCRL2 actions. μ++ might be able to borrow these schemes such that we can also use μ++ to write requirements for these higher level languages, rather than only mCRL2.

Bibliography

[1] Anders Sandberg and Nick Bostrom. Whole Brain Emulation: A Roadmap. Technical report, Future of Humanity Institute, Oxford University, 2008. Appendix B.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A View of the Parallel Computing Landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[3] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

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

[5] E. M. Maximilien and L. Williams. Assessing test-driven development at IBM. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 564–569, May 2003.

[6] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996.

[7] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sep. 1990.

[8] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[9] Daniela Remenska, Tim A. C. Willemse, Jeff Templon, Kees Verstoep, and Henri Bal. Property Specification Made Easy: Harnessing the Power of Model Checking in UML Designs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 17–32, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[10] Daniela Remenska. Bringing Model Checking Closer To Practical Software Engineering. Master's thesis, Vrije Universiteit Amsterdam, 2016.

[11] Yi Ling Hwong, Jeroen J.A. Keiren, Vincent J.J. Kusters, Sander Leemans, and Tim A.C. Willemse. Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. *Science of Computer Programming*, 78(12):2435 – 2452, 2013. Special

Section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011).

[12] T. C. Willemse, W. Fokkink, J. Templon, K. Verstoep, H. Bal, and D. Remenska. Using Model Checking to Analyze the System Behavior of the LHC Production Grid. In *Cluster Computing and the Grid, IEEE International Symposium on*, pages 335–343, Los Alamitos, CA, USA, may 2012. IEEE Computer Society.

[13] N.J.M. Nieuwelaar, van den. *Supervisory machine control by predictive-reactive scheduling*. PhD thesis, Department of Mechanical Engineering, 2004.

[14] R.J.W. Jonk. *The semantics of ALIAS defined in mCRL2*. PhD thesis, Department of Mathematics and Computer Science, 2016.

[15] L. L. F. Merkx, P. J. L. Cuijpers, and H. M. Duringhof. Algebraic Software Analysis and Embedded Simulation of a Driving Robot. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, pages 473–480, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[16] Radu Mateescu. Property Pattern Mappings for Regular Alternation-Free μ-Calculus. URL `http://www.inrialpes.fr/vasy/cadp/resources/evaluator/rafmc.html`, 2019. [Online; accessed 17-January-2019].

[17] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. The Patterns. URL `http://patterns.projects.cs.ksu.edu/documentation/patterns.shtml`, 1970. [Online; accessed 23-January-2019].

[18] Daniela Remenska. PASS/PatternMuCalculusFormat.java at master · remenska/PASS · GitHub. URL `https://github.com/remenska/PASS/blob/master/src/info.remenska.PASS/src/info/remenska/PASS/wizards/PatternMuCalculusFormat.java`, 2013. [Online; accessed 23-January-2019].

[19] Bettina Laugwitz, Theo Held, and Martin Schrepp. Construction and evaluation of a user experience questionnaire. volume 5298, pages 63–76, 11 2008.

[20] Martin Schrepp, Andreas Hinderks, and Jörg Thomaschewski. Construction of a Benchmark for the User Experience Questionnaire (UEQ). *International Journal of Interactive Multimedia and Artificial Intelligence*, 4:40–44, 06 2017.

Appendix A

Modal μ-calculus formulae

A.1 Syntax of modal μ-calculus formulae

The modal μ-calculus formulae that we describe in this document are a propositional modal logic extended with fixed point operators. The official implementation as used in the mCRL2 framework uses a notion of time as well, but we will forego that in this document for brevity. This logic is based on actions α; these actions are either an unnamed action τ, a named action a, or a multiaction $\alpha|\alpha$. Colloquially, multiactions are understood to be actions that happen simultaneously. Named actions can have data assigned to them—such data will further distinguish these actions from those similarly named, but with different data. Actions with data use notation $a(t, \ldots, t)$. These actions thus have the following syntax;

$$\alpha ::= \tau \mid a(t, \ldots, t) \mid \alpha|\alpha$$

Where t has the following syntax (with x a variable name and f a function name);

$$\begin{aligned} t ::= {} & x \mid f \mid [] \mid \{\} \mid \{:\}[t, \ldots, t] \mid \{t, \ldots, t\} \mid \{t : t, \ldots, t : t\} \mid \{t{:}D \mid t\} \mid \\ & \lambda x_1{:}D_1, \ldots, x_n{:}D_n \,.\, t \mid t(t, \ldots, t) \mid \forall x{:}D \,.\, (t) \mid \exists x{:}D \,.\, (t) \mid \\ & t \ \textbf{whr}\ x_1 = t, \ldots, x_n = t \ \textbf{end} \end{aligned}$$

We can also have expressions based on these data. These expressions should resolve to a boolean value; passing badly typed expressions results in undefined behavior (the mCRL2 toolkit will liberally attempt to find a reasonable typing). The following is only a partial syntax;

$$\begin{aligned} b ::= {} & t \mid \textbf{true} \mid \textbf{false} \mid \neg b \mid b \wedge b \mid b \vee b \mid b \rightarrow b \mid b = b \mid b \neq b \mid t < t \mid t \leq t \mid \\ & t \geq t \mid t > t \mid t + t \mid t - t \mid t \times t \mid t \div t \mid t \ \mathsf{div}\ t \mid t \ \mathsf{mod}\ t \mid -t \mid t \in t \mid \\ & \forall d{:}D \,.\, (b) \mid \exists d{:}D \,.\, (b) \mid \ldots \end{aligned}$$

Actions can be collected into sets, so-called action formulae, to simplify the notation of many μ-

formulae. More specifically, sets allow for quantification over data. Additionally, we can use concepts like **true** to denote the set of all actions. The syntax of such action formulae is;

$$af ::= b \mid \textbf{true} \mid \textbf{false} \mid \alpha \mid \overline{af} \mid af \cap af \mid af \cup af \mid \forall d{:}D\,.\,(af) \mid \exists d{:}D\,.\,(af)$$

We would also like to describe sequences of actions. For this, regular expressions are used;

$$R ::= \varepsilon \mid af \mid R \cdot R \mid R + R \mid R^{\star} \mid R^{+}$$

And finally, the logic to provide the propositions to reason about;

$$\begin{aligned}\phi ::= {} & b \mid \textbf{true} \mid \textbf{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \forall d{:}D\,.\,(\phi) \mid \exists d{:}D\,.\,(\phi) \mid \\ & \langle R\rangle\,\phi \mid [R]\,\phi \mid \mu X(d_1 : D_1 \leftarrow t_1,\, \ldots)\,.\,(\phi) \mid \nu X(d_1 : D_1 \leftarrow t_1,\, \ldots)\,.\,(\phi) \mid \\ & X(t_1,\, \ldots)\end{aligned}$$

A.2 Semantics of modal μ-calculus formulae

Definition 8 (Semantics of action formulae)**.** Let $\mathcal{D} = (\Sigma, E)$ be a data specification with $\Sigma = (\mathcal{S}, \mathcal{C}_{\mathcal{S}}, \mathcal{M}_{\mathcal{S}})$ a signature, $\mathcal{A} = \{M_D \mid D \in \mathcal{S}\}$ a $\mathcal{D}$-structure, $[\![\cdot]\!]$ a $\mathcal{D}$-model, and $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$ a timed transition system where Act consists of all semantical multi-actions a.

Let af be an action expression. We define the interpretation of af, notation $[\![af]\!]^{\sigma}$ where σ is a valuation, as a set of semantical multi-actions, inductively by;

$$[\![\textbf{true}]\!]^{\sigma} = \textit{Act} \tag{A.1}$$

$$[\![\textbf{false}]\!]^{\sigma} = \emptyset \tag{A.2}$$

$$[\![b]\!]^{\sigma} = \begin{cases} \textit{Act} & \text{if } [\![\sigma(b)]\!] = \textbf{true} \\ \emptyset & \text{if } [\![\sigma(b)]\!] = \textbf{false} \end{cases} \tag{A.3}$$

$$[\![\alpha]\!]^{\sigma} = [\![\sigma(\alpha)]\!] \tag{A.4}$$

$$[\![\overline{af}]\!]^{\sigma} = \textit{Act} \setminus [\![af]\!]^{\sigma} \tag{A.5}$$

$$[\![af_1 \cap af_2]\!]^{\sigma} = [\![af_1]\!]^{\sigma} \cap [\![af_2]\!]^{\sigma} \tag{A.6}$$

$$[\![af_1 \cup af_2]\!]^{\sigma} = [\![af_1]\!]^{\sigma} \cup [\![af_2]\!]^{\sigma} \tag{A.7}$$

$$[\![\forall d{:}D\,.\,(af)]\!]^{\sigma} = \bigcap_{d \in M_D} [\![af]\!]^{\sigma[d/x]} \tag{A.8}$$

$$[\![\exists d{:}D\,.\,(af)]\!]^{\sigma} = \bigcup_{d \in M_D} [\![af]\!]^{\sigma[d/x]} \tag{A.9}$$

Definition 9 (Semantics of modal formulae)**.** Let $\mathcal{D} = (\Sigma, E)$ be a data specification with $\Sigma = (\mathcal{S}, \mathcal{C}_{\mathcal{S}}, \mathcal{M}_{\mathcal{S}})$ a signature, $\mathcal{A} = \{M_D \mid D \in \mathcal{S}\}$ a $\mathcal{D}$-structure, $[\![\cdot]\!]$ a $\mathcal{D}$-model, and $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$ a timed transition system where Act consists of all semantical multi-actions a.

Let ϕ be a modal formula. We inductively define the interpretation of ϕ, notation $[\![\phi]\!]^{\sigma,\rho}$ where σ is a valuation and ρ is a logical variable valuation, as a set of states where ϕ is valid, by;

$$[\![\textbf{true}]\!]^{\sigma,\rho} = S \tag{A.10}$$

$$[\![\textbf{false}]\!]^{\sigma,\rho} = \emptyset \tag{A.11}$$

$$[\![b]\!]^{\sigma,\rho} = \begin{cases} S & \text{if } [\![\sigma(b)]\!] = \textbf{true} \\ \emptyset & \text{if } [\![\sigma(b)]\!] = \textbf{false} \end{cases} \tag{A.12}$$

$$[\![\neg\phi]\!]^{\sigma,\rho} = S \setminus [\![\phi]\!]^{\sigma,\rho} \tag{A.13}$$

$$[\![\phi_1 \wedge \phi_2]\!]^{\sigma,\rho} = [\![\phi_1]\!]^{\sigma,\rho} \cap [\![\phi_2]\!]^{\sigma,\rho} \tag{A.14}$$

$$[\![\phi_1 \vee \phi_2]\!]^{\sigma,\rho} = [\![\phi_1]\!]^{\sigma,\rho} \cup [\![\phi_2]\!]^{\sigma,\rho} \tag{A.15}$$

$$\begin{aligned}[\![\langle af\rangle\,\phi]\!]^{\sigma,\rho} = \{\, & s \in S \mid \exists\, s' \in S,\ \alpha \in [\![af]\!]^{\sigma} : \\ & s \xrightarrow{\alpha} s' \wedge s' \in [\![\phi]\!]^{\sigma,\rho} \,\}\end{aligned} \tag{A.16}$$

$$\begin{aligned}[\![[af]\,\phi]\!]^{\sigma,\rho} = \{\, & s \in S \mid \forall\, s' \in S,\ \alpha \in [\![af]\!]^{\sigma} : \\ & s \xrightarrow{\alpha} s' \Rightarrow s' \in [\![\phi]\!]^{\sigma,\rho} \,\}\end{aligned} \tag{A.17}$$

$$[\![\forall x{:}D \,.\, (\phi)]\!]^{\sigma,\rho} = \bigcap\nolimits_{d \in M_D} [\![\phi]\!]^{\sigma[d/x],\rho} \tag{A.18}$$

$$[\![\exists x{:}D \,.\, (\phi)]\!]^{\sigma,\rho} = \bigcup\nolimits_{d \in M_D} [\![\phi]\!]^{\sigma[d/x],\rho} \tag{A.19}$$

$$\begin{aligned}[\![\mu X(x{:}D \leftarrow t) \,.\, (\phi)]\!]^{\sigma,\rho} = \bigcap\nolimits_{f \in M_D \to 2^S} \{\, f([\![t]\!]^{\sigma}) \mid \forall\, d \in M_D \;:\; \\ f(d) = [\![\phi]\!]^{\sigma[d/x],\rho[X \leftarrow f]} \}) \end{aligned} \tag{A.20}$$

$$\begin{aligned}[\![\nu X(x{:}D \leftarrow t) \,.\, (\phi)]\!]^{\sigma,\rho} = \bigcup\nolimits_{f \in M_D \to 2^S} \{\, f([\![t]\!]^{\sigma}) \mid \forall\, d \in M_D \;:\; \\ f(d) = [\![\phi]\!]^{\sigma[d/x],\rho[X \leftarrow f]} \}) \end{aligned} \tag{A.21}$$

$$[\![X(t)]\!]^{\sigma,\rho} = \rho(X)([\![t]\!]^{\sigma}) \tag{A.22}$$

We say that ϕ holds in A iff $s_0 \in [\![\phi]\!]^{\sigma,\rho}$ for any σ, ρ, and $[\![\cdot]\!]$.

Definition 10 (Equivalence of modal formulae)**.** Let $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ be a process specification and let ϕ be a modal formula. We say that ϕ is valid in PS iff for any $\mathcal{D}$-structure $\mathcal{A}$, $\mathcal{D}$-model $[\![\cdot]\!]$, and timed transition system A (that is; the semantics of PS given $\mathcal{A}$ and σ), ϕ holds in A.

We say that two modal formulae ϕ and ψ are equivalent iff for all process specifications, ϕ is valid iff ψ is valid.

A.3 Relation between symbols

In section A.2, where the semantics of modal μ-formulae is described, we do not describe much of the syntax we have listed in section A.1. That is because there is plenty of overlap in the definitions of the syntax, even when we do not consider dualities. Most notably, we do not need regular expressions at all (and do not semantically define them). In the syntax, we are allowed to place regular expressions in the box and diamond modalities. We can rewrite all possible regular expression structures as follows;

$$\begin{array}{rlrl}
[\varepsilon]\,\phi \vdash & [\textbf{false}^\star]\,\phi & \langle\varepsilon\rangle\,\phi \vdash & \langle\textbf{false}^\star\rangle\,\phi \\
[R_1 \cdot R_2]\,\phi \vdash & [R_1]\,[R_2]\,\phi & \langle R_1 \cdot R_2\rangle\,\phi \vdash & \langle R_1\rangle\langle R_2\rangle\,\phi \\
[R_1 + R_2]\,\phi \vdash & [R_1]\,\phi \wedge [R_2]\,\phi & \langle R_1 + R_2\rangle\,\phi \vdash & \langle R_1\rangle\,\phi \vee \langle R_2\rangle\,\phi \\
[R^\star]\,\phi \vdash & \nu X\,.\,(\phi \wedge [R]\,X) & \langle R^\star\rangle\,\phi \vdash & \mu X\,.\,(\phi \vee \langle R\rangle\,X) \\
[R^+]\,\phi \vdash & [R \cdot R^\star]\,\phi & \langle R^+\rangle\,\phi \vdash & \langle R \cdot R^\star\rangle\,\phi
\end{array}$$

Finally, we miss the conditional structure $\phi_1 \rightarrow \phi_2$, but we know by material implication that we may rewrite such structures to $\neg\phi_1 \vee \phi_2$. Therefore, we can mechanically rewrite every syntactically valid statement (given that Boolean expressions are indeed Boolean) to a semantically interpretable statement.