Program Matching with Semantic Patterns

Version of April 8, 2025

Pepijn Vunderink

Program Matching with Semantic Patterns

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Pepijn Vunderink
born in Den Haag, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Program Matching with Semantic Patterns

Author: Pepijn Vunderink
Student id: 4913841

Abstract

Current tools for pattern matching computer programs often operate on abstract syntax trees or other static representations of programs. These approaches, though efficient, are fundamentally limited when it comes to capturing the dynamic behavior of programs. For example, it is not always possible to express (concise) syntactic patterns that capture programs which are semantically equivalent but differ in their syntactic representation. A tool that takes into account the behavior (or dynamic semantics) of programs would be able to capture programs that are semantically equivalent in a more concise manner with a single pattern. Additionally, taking into account program behavior leads to more precise pattern matching, by excluding unreachable paths of computation.

In this thesis, we explore a novel method, based on behavioral models of programs, that allows patterns to take into account the dynamic semantics of a program. We propose the Dyno pattern language, in which concrete object language syntax can be used to express intuitive semantic patterns of programs. Pattern matching is performed by translating Dyno patterns to μ-calculus formulas and model checking these formulas against models extracted from object programs.

Because our method is based on dynamic models of programs, we are fundamentally limited by the halting problem. In favor of precision, our method compromises on efficiency and termination guarantees. In particular, termination is not guaranteed when the extracted model of a program has infinitely many states. To recover termination in some cases, we provide the facility to express bounds on input parameters, limiting the search space while compromising on soundness.

We recognize some limitations in our work, including a lack of match evidence (e.g. the location of a match in the object program's syntax tree), as well as holes in Dyno's expressiveness. To address the latter issue, we suggest operators that could be added to Dyno in the future.

Thesis Committee:

Chair:	Dr. J.G.H. (Jesper) Cockx, Faculty EEMCS, TU Delft
Committee Member:	Dr. B. (Burcu) Özkan, Faculty EEMCS, TU Delft
Committee Member:	Ir. L. (Luka) Miljak, Faculty EEMCS, TU Delft
University Supervisor:	Ir. L. (Luka) Miljak, Faculty EEMCS, TU Delft

Preface

The journey of this thesis began with a project under the supervision of Casper Bach Poulsen, focused on developing a Haskell-based clone of the program transformation tool Coccinelle. After completion of the project, I asked Casper if he would be interested in supervising my master thesis, and if he had a topic in mind. He expressed interest in formalizing program transformations using modal μ-calculus. Together with Luka Miljak, one of Casper's PhD students, we proceeded with initial exploration of this topic. Shortly after, Casper had to abandon his supervisory role in this project due to moving back to Denmark with his family, which meant Luka would be taking over the main supervision of my thesis. With Casper gone, we struggled continuing the project and decided to shift to a new topic. Luka suggested it might be interesting to focus on pattern matching, a precursor to program transformation, and in particular explore pattern matching based on the dynamic semantics (i.e. behavior) of programs. I agreed, and thus the definitive topic for my thesis was established.

Research is never conducted in a vacuum. Many people have contributed to this thesis, either directly or indirectly. I would like to express my gratitude to everyone involved in this project, with special thanks to those who played a particularly significant role:

I am grateful to Luka for offering invaluable feedback throughout this project, and providing guidance during our weekly meetings. I also sincerely appreciate the detailed feedback from Jesper on multiple drafts of this thesis, including the suggestions he himself referred to as 'nitpicks.' Their combined insights have been instrumental in shaping this thesis into its current form. I thank Flip and Jan Friso for expressing interest in my work and guiding me through the ins and outs of the mCRL2 toolset. I am grateful for the love and support of my parents, not only during my work on this thesis, but throughout the entirety of my studies and everything that came before. Even – or perhaps *especially* – during tougher times I could count on their support, and for that I am forever grateful. Furthermore, I thank my roommates for putting up with my prickly nature during stressful times. I thank Fernão for cooking an extra time on my behalf during particularly busy weeks. I thank Roeland and Reinier for their enthusiasm and waiting patiently for a copy of this thesis, even after many inquiries.

Finally, I extend my gratitude to you, the reader, for engaging with this thesis.

Pepijn Vunderink
Delft, the Netherlands
April 8, 2025

Contents

Chapter 1

Introduction

Program matching is the process of searching for code fragments or subprograms that adhere to a specific pattern. Applications of program matching include program comprehension, bug detection, optimization and refactoring. A rudimentary form of program matching is plain-text search, which is an indispensable feature in any integrated development environment and even simple text editors typically support it. In these same tools, more often than not it is possible to specify syntactic patterns in the form of regular expressions, as a means to capture code fragments more flexibly. Still, one needs to be mindful of details such as comments, whitespace and formatting in general. A next step would be search based on abstract syntax trees (ASTs), to match code more precisely and take code formatting out of the equation.

There are situations, however, where approaches based merely on (abstract) syntax are simply inadequate. What if, for example, one wants to find code that spans multiple function bodies or some behavioral pattern that occurs in a program, regardless of its precise syntactic form? As an example, say we want to find occurrences of a use-after-free bug, where a resource is accessed that has previously been invalidated. Such a query could be expressed with the following pseudo-pattern:

Example 1.1. A pattern expressing a use-after-free bug.

```
1 free(@m)
2 ...[!alloc(@m)]
3 access(@m)
```

`free(@m)` captures the freeing of resource `@m`, `...[!alloc(@m)]` matches arbitrary code sequences that are not (re-)allocations of `@m` and `access(@m)` captures accesses to resouce `@m`. Such a query is hard to encode in a sound manner using purely syntax-based pattern matching, because its occurrence can take on many, distinct, syntactic forms and trying to capture all possibilities would result in an impractically large pattern.

As another example, consider the following pattern for searching infinite loops.

Example 1.2. A pattern that finds loops of which the condition always evaluates to true. In this pattern, `@e->true` means "an arbitrary expression that evaluates to true."

```
1 ...
2 while(@e->true) {
3   ...
4 }
```

Expressing the above pattern requires reasoning about the runtime value of the loop condition, which is beyond the capabilities of purely syntactic pattern matching tools.

Lastly, consider a scenario where we aim to find an expression that is semantically equivalent to a given formula, without worrying about its precise syntactic representation. For instance, take the following pattern:

Example 1.3. `!@e || @e'`

The goal is to match all expressions that are semantically equivalent to this pattern. For example, the following expressions should all match:

- `!x || y`
- `x || !y`
- `!(x && !y)`

Using a purely syntactic pattern matching tool, one would have to enumerate all possible syntactic variations. This is not only cumbersome but becomes infeasible when there are infinitely many equivalent forms, as is the case in this example.

To express the previous examples concisely, we must move beyond syntactic pattern matching. A key insight is that these examples depend on the *behavior* (or *dynamic semantics*) of a program. Thus, a pattern matching tool capable of expressing such patterns must be able to reason about program behavior.

This thesis proposes a new pattern matching language: Dyno, designed to express behavioral patterns to match against the fictional imperative-style programming language Imp. While preserving the ability to match programs syntactically, Dyno matches against behavioral models of programs, instead of syntax trees or other static representations of programs.

Using this new language, a pattern such as the aforementioned use-after-free pattern can be expressed concisely and captures occurrences of the use-after-free bug more precisely – reporting fewer false positives – than syntactic or static pattern matching could.

Accounting for dynamic semantics inherently comes with limitations, however. Unless the object language is completely trivial, it is not possible to match arbitrary semantic patterns against arbitrary programs in finite time. In general, semantic properties of programs are undecidable, as a result of Rice's theorem (Rice 1953). An implication of this is that we cannot dynamically match on execution paths of arbitrary programs while preserving termination. In our case, the problem originates from the use of behavioral models of programs that represent *all* possible executions of a program. Sometimes, such behavioral models are infinitely large. To recover termination in the scenario where unknown input parameters cause the behavioral model to be infinite, we provide the ability to bound input parameters of a program. When input parameters are bounded, the behavioral model only represents a subset of all possible executions, implying a compromise on soundness. Unbounded input parameters are not the only cause of infinitely large behavioral models. For instance, a non-converging infinite loop in a program also causes the resulting behavioral model to be infinitely large. We do not guarantee termination of pattern matching in such scenarios.

We implement Dyno as a translation to μ-calculus formulas. Pattern matching is performed by model checking the resulting formulas against the behavioral models of Imp programs. The translation to μ-calculus doubles as a definition of Dyno's semantics. The pipeline of our implementation is best summarized by the diagram in Figure 1.1.

Concretely, the main contributions of this thesis are as follows.

- In Section 3.2 we introduce a new pattern matching language (Dyno) that is designed to express behavioral patterns of Imp programs, while supporting concrete object language syntax.

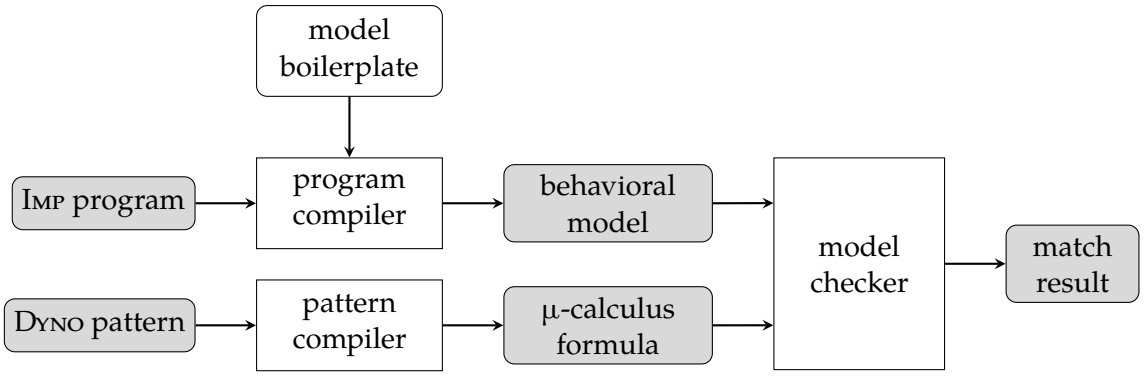


Figure 1.1. An overview of how Dyno patterns are compiled and matched against an input program. The output is a boolean value indicating whether the input pattern matches the input program.

- In Section 3.3 we demonstrate Dyno's expressiveness by discussing examples of practically applicable patterns.
- We define a method for extracting behavioral models from Imp programs in Chapter 4, which is implemented as a process specification using the mCRL2 toolset (Bunte et al. 2019).
- We provide a translation of Dyno into μ-calculus formulas in Chapter 5. This translation not only serves as an implementation but also as a formal semantic definition of Dyno.
- We provide a full implementation of the compilation pipeline shown in Figure 1.1 using the Spoofax language workbench (Kats and Visser 2010). This implementation is discussed in Section 5.3.
- We validate the correctness of our implementation of Dyno using an automated test suite, as discussed in Section 5.3.1.
- In Chapter 6 we discuss limitations in our approach and the current state of Dyno. Notably, we identify holes in expressiveness, which we suggest fixing in Section 6.3, by adding new operators to the language.

The remainder of this thesis is laid out as follows. In Chapter 2 we formally introduce the modal μ-calculus, labeled transition systems (LTSs) and the mCRL2 specification language. Chapter 3 commences with a definition of the object language (Imp) that we design our pattern language around in Section 3.1. In Section 3.2 we introduce the Dyno pattern language and explain each of its operators with examples, continuing with some practical examples in Section 3.3. We then show how to extract behavioral models from Imp programs using an mCRL2 specification in Sections 4.1 and 4.2 and how to extend this model with syntactic information of internal computation in Section 4.3 to facilitate pattern matching. In Chapter 5 we discuss the translation of Dyno into μ-calculus formulas, starting with examples for each operator in Section 5.1, followed up by a complete overview in Section 5.2. We briefly discuss the Spoofax implementation of Dyno and Imp compilation in Section 5.3, including an overview of the test suite used to verify its correctness. We discuss limitations of our work and how Dyno can be improved upon in future work in Chapter 6. In Chapter 7 work related to this thesis is discussed. The thesis concludes with a brief summary and closing remarks in Chapter 8.

Chapter 2

Modal μ-calculus and mCRL2

An important goal of this thesis is to make use of existing software to perform pattern matching. Not only does this relieve us of the burden of having to develop a completely new algorithm for pattern matching, but it also allows us to relate the problem of semantic pattern matching to a well-studied logic: the modal μ-calculus.

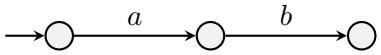
This chapter offers an introduction to labeled transition systems (LTSs), the modal μ-calculus, as well as the mCRL2 specification language. Although we go into some depth, we focus on the theory required for this thesis. For a more comprehensive introduction to these topics, the book Modeling and Analysis of Communicating Systems is advised (Groote and Mousavi 2014), of which in particular chapters 2, 3, 4, 6 & 15 are relevant to this thesis.

We start by introducing labeled transition systems in Section 2.1. In Section 2.2 we introduce Hennessy-Milner logic (HML), a precursor to modal μ-calculus. We show that by extending HML with fixed point operators we arrive at the modal μ-calculus in Section 2.3. Section 2.4 contains an introductory tutorial on how to model processes in mCRL2, using custom data types, rewrite rules and process equations. We return to the μ-calculus in Section 2.5, where we extend the μ-calculus to be able to deal with the data that we define in mCRL2 specifications.

2.1 Labeled transition systems

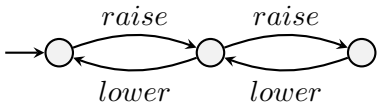
To facilitate pattern matching we model object programs using labeled transition systems (LTSs). LTSs are directed graphs with labeled edges and they can be used to model system behavior (Groote and Mousavi 2014). An example of an LTS with three states is depicted below.

Example 2.1. A simple LTS with three states and two actions: a and b. The left-most state is the initial state, indicated by the incoming arrow.



LTSs can be used to model systems. In particular, they model the possible states that a system can be in, as well as what actions are possible in each state and how they lead to new states. Consider the following example of a platform that can be raised and lowered.

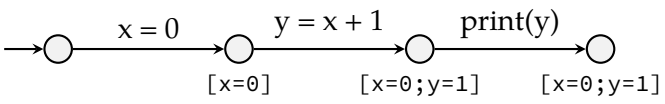
Example 2.2. An LTS modeling a platform with two actions: $lower$ and $raise$. The platform has 3 levels (modeled as states): low, mid and high.



We are interested in modeling programs. To illustrate one might represent a program using an LTS, consider the following sequential program:

```
1 x = 0;
2 y = x + 1;
3 print(y)
```

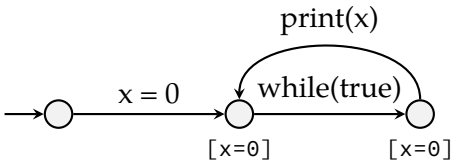
Which we encode as follows:



The LTS nicely reflects the linear nature of this program. We have annotated the states with their environments, which are the mappings of variables to values. These annotations are normally not part of an LTS, but they provide an intuition to what the states might represent: the state of a program after having taken a trail of actions to get there. In this case "taking an action" might be interpreted as evaluating an expression. An example of a program that results in non-linear behavior would be a program with a loop, e.g.:

```
1 x = 0;
2 while (true) {
3   print(x);
4 }
```

An LTS that represents this program might look like this:



Having seen some examples of labeled transition systems, let us now give their formal definition.

Definition 2.1. A *labeled transition system* (LTS) is a tuple $\mathcal{M} = (S, Act, \{\xrightarrow{a}\}_{a \in Act}, s_0)$ where

- S is a set of *states*.
- Act is a set of *actions*.
- $\forall a \in Act.\ \xrightarrow{a} \subseteq S \times S$ is a *transition relation* describing which states are connected by action a. We use $s \xrightarrow{a} s'$ as shorthand for $(s, s') \in \xrightarrow{a}$.
- $s_0 \in S$ is the *initial state*.

Knowing what LTSs are and that we can use them to represent programs, let us remind ourselves of what we *actually* wanted to do: perform pattern matching on programs. To this end we need to be able to reason about LTSs of programs. There happens to be a logical framework that, among other things, allows us to reason about LTSs. This framework is called modal logic. We are going to use a specific modal logic, the modal μ-calculus, to reason about the programs that we encode as LTSs. First we discuss a simpler modal logic, after which we build up to the more expressive μ-calculus.

2.2 Hennessy-Milner logic

Hennessy-Milner logic (HML) is a modal logic introduced by Hennessy and Milner (1980). As we will see in the next section, it is a simpler and less expressive subset of modal μ-calculus. Both logics are part of a larger family of modal logics. Modal logics, according to Blackburn and Benthem (2007) have been developed as tools for reasoning about time, beliefs, computational systems and necessity and possibility. But more important for our purpose is that modal logics can be thought of as logics for reasoning about labeled transition systems. While no prior knowledge of modal logics is assumed, to be able to follow the coming sections it is helpful to have a basic understanding of first-order logic.

Now, let us define the syntax of Hennessy-Milner logic formulas.

Definition 2.2. Syntax of HML formulas. Here a represents actions from the action set of some LTS: $a \in Act$.

$$
\begin{array}{llll}
\text{HML} & \phi ::= & \top & \text{constant (true)} \\
& \mid & \bot & \text{constant (false)} \\
& \mid & \neg\phi & \text{negation} \\
& \mid & \phi \vee \phi & \text{disjunction} \\
& \mid & \phi \wedge \phi & \text{conjunction} \\
& \mid & \langle a \rangle \phi & \text{diamond} \\
& \mid & [a]\phi & \text{box}
\end{array}
$$

Just like propositional logic, HML has negation, disjunction and conjunction. What sets HML apart are the box and diamond operators. These are so-called *modality* operators. They allow us to reason about transitions and what happens after a transition. The diamond modality, $\langle a \rangle \phi$, informally means "there is a transition a to some state where ϕ holds." Whereas the box modality, $[a]\phi$, means "all a-transitions should result in a state where ϕ holds."

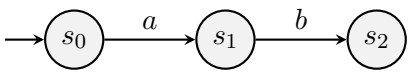
As promised, we can use HML formulas to reason about LTSs. In fact the semantics of a HML formula are usually defined over them. We give the semantics of HML in terms of the set of states in which a formula holds.

Definition 2.3. Given an arbitrary LTS $\mathcal{M} = (S, Act, \{\xrightarrow{a}\}_{a \in \text{Act}}, s_0)$, the semantics of HML formulas are defined by the equations below. The notation $[\![\phi]\!]$ means "the states where formula ϕ holds." We consider formula ϕ valid for an LTS if it is valid in its initial state, i.e. $s_0 \in [\![\phi]\!]$.

$$
\begin{aligned}
[\![\top]\!] &= S \\
[\![\bot]\!] &= \varnothing \\
[\![\neg\phi]\!] &= S \backslash [\![\phi]\!] \\
[\![\phi_1 \wedge \phi_2]\!] &= [\![\phi_1]\!] \cap [\![\phi_2]\!] \\
[\![\phi_1 \vee \phi_2]\!] &= [\![\phi_1]\!] \cup [\![\phi_2]\!] \\
[\![\langle a \rangle \phi]\!] &= \{s \in S \mid \exists s' \in S \text{ s.t. } s \xrightarrow{a} s' \text{ and } s' \in [\![\phi]\!]\} \\
[\![[a]\phi]\!] &= \{s \in S \mid \forall s' \in S \text{ s.t. } s \xrightarrow{a} s' \text{ implies } s' \in [\![\phi]\!]\}
\end{aligned}
$$

Let us have a look at an example. Consider the formula $\langle a \rangle \top$, which means "there is an a-action that leads to a state where $\top$ holds". The following is an example of an LTS where this formula holds.

Example 2.3. Example of an LTS where $\langle a \rangle \top$ holds.



To build intuition we give some more examples of formulas, together with their informal meanings and the states where they hold in the LTS of Example 2.3. The reader is encouraged to verify the following examples:

- $\neg\langle b\rangle\top$: "There is no b transition." (s_0,s_2)
- $[b]\bot$: "There is no b transition." (s_0, s_2)
- $[a]\langle b\rangle\top$: "After all a transitions, a b transition follows." (s_0, s_1, s_2)
- $[a]\bot \wedge [b]\bot$: "There are no a transitions and no b transitions." (s_2)

The first two formulas reveal an interesting correspondence between the box and diamond modalities, indeed the examples have the exact same meaning. This correspondence can be attributed to the duality between the two modalities. It turns out we can express one in terms of the other (and vice versa): $\langle a\rangle\phi = \neg[a]\neg\phi$.

It is often useful to be able to reason about more than one action at the same time. For this we introduce new syntax that allows sets of actions to be used within modality operators.

Definition 2.4. Syntax of action formulas.

ActionFormula	α	::=	a	regular action
		|	$\top$	any action
		|	$\bot$	no action
		|	$\overline{\alpha}$	complement
		|	$\alpha \cup \alpha$	union
		|	$\alpha \cap \alpha$	intersection

Action formulas should be interpreted as sets of actions. For example, $\top$ represents all actions in the action set of some LTS, while $\bot$ represents the empty set. We formalize this notion by defining the semantics of action formulas as a translation to sets.

Definition 2.5. Semantics of action formulas. Using double bracket notation ($[\![\cdot]\!]_{Act}$), rules for translating action formulas into sets of actions are defined as follows. Act represents the action set of a given LTS.

$$
\begin{aligned}
[\![a]\!]_{Act} &= \{a\} \\
[\![\top]\!]_{Act} &= Act \\
[\![\bot]\!]_{Act} &= \varnothing \\
[\![\overline{\alpha}]\!]_{Act} &= Act \backslash [\![\alpha]\!]_{Act} \\
[\![\alpha_1 \cup \alpha_2]\!]_{Act} &= [\![\alpha_1]\!]_{Act} \cup [\![\alpha_2]\!]_{Act} \\
[\![\alpha_1 \cap \alpha_2]\!]_{Act} &= [\![\alpha_1]\!]_{Act} \cap [\![\alpha_2]\!]_{Act}
\end{aligned}
$$

As an example, the HML formula $\langle\top\rangle\top$, means "there is an action that leads to a state where $\top$ holds," or more briefly "there is an action." Another example is $[\top]\bot$, meaning "all actions lead to a state where $\bot$ holds," in other words "there is no action."

Instead of redefining HML, we show how to desugar modalities with action formulas into the core HML syntax defined in Definition 2.2.

Definition 2.6. Desugaring action formula modalities. Using double bracket notation ($[\![\cdot]\!]_{Act}$), rules for translating action formulas into sets of actions are defined as follows. Act represents the action set of a given LTS.

$$
\begin{aligned}
[\![\langle\alpha\rangle\phi]\!]_{Act} &= \bigvee_{a \in [\![\alpha]\!]_{Act}} \langle a\rangle\phi \\
[\![[\alpha]\phi]\!]_{Act} &= \bigwedge_{a \in [\![\alpha]\!]_{Act}} [a]\phi
\end{aligned}
$$

To illustrate how the desugaring works, consider an LTS with two actions: $Act = \{\mathsf{a}, \mathsf{b}\}$. Then desugaring $\langle\top\rangle\phi$ goes as follows:

$$\begin{aligned}[\![\langle\top\rangle\phi]\!]_{Act} &= \bigvee_{a \in [\![\top]\!]_{Act}} \langle a\rangle\phi \\ &= \bigvee_{a \in Act} \langle a\rangle\phi \\ &= \langle\mathsf{a}\rangle\phi \vee \langle\mathsf{b}\rangle\phi\end{aligned}$$

Intermezzo. *A different way to look at HML.* We have described HML as a language for expressing properties about LTSs. From a different perspective, HML can be thought of as a basis for *characterizing* LTSs. By looking at the sets of HML formulas that are true for different LTSs, we can compare their meanings. In particular, if the same formulas hold for two LTSs, they are considered equivalent through the lens of HML. This perspective of providing a notion of equivalence is close to the original purpose of HML, as Hennessy and Milner developed HML to reason about non-deterministic and concurrent programs and equivalence between them. This perspective is valid for any modal logic, not just HML, and different modal logics result in different notions of equivalence. Although we will not use this principle directly, it does relate to the application of pattern matching, where we are interested in (sub)programs that are equivalent with respect to some pattern. This might give insight into why translating patterns into modal formulas to perform pattern matching is not such a wild idea.

While HML is a great starting point, it does lack expressivity and flexibility. We might want to express that something will happen in the future, while not knowing exactly how many, or even which, actions it takes to get there. If we allowed regular expressions as actions, for example, we could express that something happens after zero or more actions: e.g. $\langle a\star\rangle\phi$ ("ϕ is true after zero or more a actions.").

2.3 The modal μ-calculus

In this section we will show that by adding *fixed point operators* to HML, we obtain a logic that is much more expressive. In particular the resulting logic allows for expressing recursion in formulas. The logic is called μ-calculus and was first introduced by Kozen (1983).

The syntax of modal μ-formulas is defined in Figure 2.1.

μ-formula	ϕ	::=	$\top$	constant (true)
		|	$\bot$	constant (false)
		|	$\neg\phi$	negation
		|	$\phi \vee \phi$	disjunction
		|	$\phi \wedge \phi$	conjunction
		|	$\langle a\rangle\phi$	diamond
		|	$[a]\phi$	box
		|	$\mu X.\phi$	least fixed point
		|	$\nu X.\phi$	greatest fixed point
		|	X	variable

Figure 2.1. Syntax of the modal μ-calculus.

Comparing this syntax to HML, we see that three syntactic additions have been made. Two new operators were added: $\mu X.\phi$ and $\nu X.\phi$, respectively the least and greatest fixed point operators. There is also syntax for referencing variables which are bound by the fixed point operators.

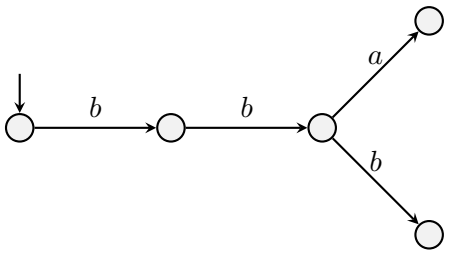
In essence, fixed points allow us to express iteration in our modal formulas. Consider the following formula as an example.

$$\mu X.\langle a\rangle\top \vee \langle b\rangle X$$

Informally, this formula expresses "there is a path of b actions that leads to an a action." If we do not care what actions are taken before the a action is reached, we could use the following formula.

$$\mu X.\langle a\rangle\top \vee \langle \top\rangle X$$

An example of an LTS where both of these formulas hold in the initial state is the following.



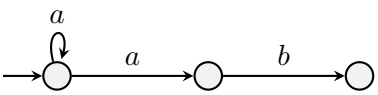
If instead we wanted to express that all subsequent paths should eventually have an a action, we could use the following formula with a box modality.

$$\mu X.\langle a\rangle\top \vee ([\top]X \wedge \langle \top\rangle\top)$$

Note that we use $\langle \top\rangle\top$ to exclude paths that end in a deadlocked state – i.e. a state without outgoing actions. This formula does not hold for the above LTS, because there is a path with only b actions that ends in a deadlocked state. The following formula without $\langle \top\rangle\top$ to exclude deadlocks does hold for the above LTS.

$$\mu X.\langle a\rangle\top \vee [\top]X$$

To understand the difference between the least fixed point (μ) and the greatest fixed point (ν), we consider the following LTS with a loop.



Now we want to answer the following question about this LTS: "is it always possible to eventually perform a b action?" We might be tempted to model this using the following least fixed point formula.

$$\mu X.\langle b\rangle\top \vee [\top]X$$

However, this formula does not hold for the above LTS. Intuitively this is because for least fixed points we are only allowed to "pass through" the fixed point variable (X) a finite amount of times. The problem is that in the above LTS, the a action can be taken infinitely many times. This means we pass through X infinitely often on the path that contains an infinite amount of a actions. If we replace the least fixed point with the greatest fixed point, we obtain a formula that *does* hold for the above LTS:

$$\nu X.\langle b\rangle\top \vee [\top]X$$

This formula holds because the greatest fixed point *can* pass through the fixed point variable an infinite amount of times. Therefore, the formula holds even on the path with infinite a actions. A caveat of the latter formula is that for LTSs with infinite loops it can ignore the left-hand side of the disjunction entirely. For example, the formula also holds for the following LTS, even though it does not have a b action at all.

To exclude such LTSs, we should additionally assert that an action b can in fact be taken eventually. Thus, to properly answer the question "is it always possible to eventually perform a b action?", we could use the following formula.

$$\nu X.\langle b\rangle\top \vee ([\top]X \wedge \mu Y.\langle b\rangle\top \vee \langle\top\rangle Y)$$

Having built an intuitive understanding of fixed point formulas, we now define their semantics formally.

Definition 2.7. Given an arbitrary LTS $\mathcal{M} = (S, Act, \{\xrightarrow{a}\}_{a\in \mathrm{Act}}, s_0)$ and a fixed point variable valuation ρ, that maps variables to sets of states, the semantics of μ-calculus formulas are defined by the equations below. The notation $[\![\phi]\!]^\rho$ means "the states where formula ϕ holds, given valuation ρ." We consider formula ϕ valid for an LTS if it is valid in its initial state, i.e. $s_0 \in [\![\phi]\!]^\rho$.

$$\begin{aligned}
[\![\top]\!]^\rho &= S \\
[\![\bot]\!]^\rho &= \varnothing \\
[\![\neg\phi]\!]^\rho &= S \backslash [\![\phi]\!]^\rho \\
[\![\phi_1 \wedge \phi_2]\!]^\rho &= [\![\phi_1]\!]^\rho \cap [\![\phi_2]\!]^\rho \\
[\![\phi_1 \vee \phi_2]\!]^\rho &= [\![\phi_1]\!]^\rho \cup [\![\phi_2]\!]^\rho \\
[\![\langle a\rangle\phi]\!]^\rho &= \{s \in S \mid \exists s' \in S \text{ s.t. } s \xrightarrow{a} s' \text{ and } s' \in [\![\phi]\!]^\rho\} \\
[\![[a]\phi]\!]^\rho &= \{s \in S \mid \forall s' \in S \text{ s.t. } s \xrightarrow{a} s' \text{ implies } s' \in [\![\phi]\!]^\rho\} \\
[\![X]\!]^\rho &= \rho(X) \\
[\![\mu X.\phi]\!]^\rho &= \bigcap_{R\subseteq S} [\![\phi]\!]^{\rho[X\mapsto R]} \\
[\![\nu X.\phi]\!]^\rho &= \bigcup_{R\subseteq S} [\![\phi]\!]^{\rho[X\mapsto R]}
\end{aligned}$$

Note that the first seven equations, defining HML formulas, are the same as in Definition 2.3 modulo passing on of valuation ρ. The last three equations define the semantics of fixed point variables, least fixed point formulas and greatest fixed point formulas, respectively.

From the semantics we can derive a relation between the least and greatest fixed point formulas. Because the greatest fixed point is defined as a union over subsets of S and the least fixed point is defined as an intersection over the same subsets of S, we know that the following relation holds.

$$[\![\mu X.\phi]\!]^\rho \subseteq [\![\nu X.\phi]\!]^\rho$$

In other words, if the least fixed point of a formula holds then the greatest fixed point of the same formula also holds. Or more succinctly put:

$$\mu X.\phi \text{ implies } \nu X.\phi$$

To conclude our introduction to the fixed point operators it should be pointed out that fixed point formulas do not always have a solution. Whenever a variable X occurs under the scope of an uneven amount of negations, as in $\mu X.\neg X$ and $\nu X.\neg([a]\neg X \vee X)$ for example, there may not be a solution. For this reason we require fixed point variables to occur under an even amount of negations, or put more formally, in all fixed point formulas $\mu X.\phi$ and $\nu X.\phi$, X must occur positively in ϕ.

2.4 Modeling processes with mCRL2

We want to model programs as LTSs to be able to verify properties about them using μ-calculus formulas. To do this we will be using the mCRL2 toolset (Bunte et al. 2019). This section provides an introduction to a subset of the mCRL2 language and is mainly intended to be used as a reference when the meaning of syntax used in Sections 4.1 to 4.3 is unclear. At minimum, to get an idea of what mCRL2 specifications look like, we encourage reading Section 2.4.1. For a more extensive overview of mCRL2, refer to Groote and Mousavi (2014). The online documentation of mCRL2 is also a helpful resource[1].

2.4.1 Process specifications

In mCRL2, we cannot specify an LTS directly. Instead, a process is described using a *process specification*. mCRL2 then allows us to verify properties about this process using μ-calculus or generate an LTS of the process to visualize it. Let us start with the following example of a process specification in mCRL2 syntax.

Example 2.4. An mCRL2 model of a platform with two actions: Raise and Lower. The platform has three possible states: Low, Mid and High. The platform starts in the Low position. The result is a process that corresponds to the LTS in Example 2.2.

```
sort Level = struct Low ? is_low | Mid ? is_mid | High ? is_high;

map raise: Level -> Level;
eqn raise(Low) = Mid;
    raise(Mid) = High;

map lower: Level -> Level;
eqn lower(Mid) = Low;
    lower(High) = Mid;

act Raise, Lower;

proc Platform(lvl: Level) =
    is_low(lvl)  -> Raise . Platform(raise(lvl))
  + is_mid(lvl)  -> (Lower . Platform(lower(lvl))
                   + Raise . Platform(raise(lvl)))
  + is_high(lvl) -> Lower . Platform(lower(lvl));

init Platform(Low);
```

The above example contains all ingredients of a process specification. It has a single sort specification, declaring the sort `Level` as a `struct` with three constructors: `Low`, `Mid` and `High`. The functions `is_low`, `is_mid` and `is_high` can be used to distinguish between these constructors.

The example also specifies two mappings from `Level` to `Level` named `lower` and `raise`. Each of these mappings have two rewrite rules associated with them that specify how to go from a `Level` to the one below or above.

Two actions are specified in this example: `Lower` and `Raise`. These actions are used in the specification of the single process defined in the example. The process is named `Platform` and has a single parameter `lvl` of sort `Level`. The values of the parameters of a process dictate

[1] https://mcrl2.org/web/user_manual/language_reference/

what state it is in. The process consists of a couple of rules that are used to determine what actions are possible in the current state. For example the first rule is:

```
is_low(lvl) -> Raise . Platform(raise(lvl))
```

This rule tells us that if the platform is currently in the `Low` state, a `Raise` action can be taken to end up in the state above `Low`, i.e. `Mid`, since `raise(Low) == Mid`. Finally, the `init Platform(Low)` line specifies that the platform starts in the `Low` state.

An overview of mCRL2 specification syntax is given in Figure 2.2

mCRL2Spec	*spec*	::=	*sort_spec spec*	sort specification
		|	*map_spec spec*	mapping
		|	*eqn_spec spec*	rewrite rule
		|	*act_spec spec*	action
		|	*proc_spec spec*	process specification
		|	`init` *proc*	initial process
	sort_spec	::=	Figure 2.4	
	map_spec	::=	Figure 2.5	
	eqn_spec	::=	Figure 2.5	
	act_spec	::=	Figure 2.6	
	proc_spec	::=	Figure 2.7	
	proc	::=	Figure 2.7	

Figure 2.2. An overview of mCRL2 specification syntax.

The next subsections explain the mCRL2 concepts we just introduced in more detail. Starting with sorts and data expressions in Section 2.4.2. Followed by rewrite rules in Section 2.4.3. Wrapping up with actions and processes in Section 2.4.5. In Section 2.5 we discuss how mCRL2 extends μ-calculus with parameterized fixed points and quantifiers to deal with data expressions. This flavor of μ-calculus can be used to verify properties about the processes defined in mCRL2 specifications. It is also the flavor of μ-calculus we use to define and implement our pattern matching language.

2.4.2 Sorts and data expressions

In mCRL2, data expressions are used to manipulate data. We define only the subset of mCRL2 data expressions used in this thesis. The syntax of data expressions is defined in Figure 2.3. For the full set of available data expressions, the mCRL2 documentation should be consulted[2].

Data expressions are strongly typed. Default types include integers, booleans, lists and functions. The usual operators on numbers and booleans are available. There are also some predefined operators on lists. The unary operator '`#`' counts elements in a list. Operator '`in`' checks for the existence of an element in a list, '`|>`' prepends an element to a list and '`++`' concatenates two lists.

Example 2.5. We can create a list using the list constructor syntax.

```
1 [0,1,2]
```

We can prepend elements to a list using the cons operator.

```
1 3 |> [0,1,2] == [3,0,1,2]
```

And we can concatenate lists.

[2]mCRL2 data expressions: https://mcrl2.org/web/user_manual/language_reference/data.html

DataExpr	e	::=	x	variable
		|	n	numeric constant
		|	`true`	true constant
		|	`false`	false constant
		|	`[`e`,...,`e`]`	list
		|	e`[`e`->`e`]`	update mapping
		|	e`(`e`,...,`e`)`	application
		|	`!`e	not
		|	`-`e	minus sign
		|	`#`e	element count
		|	`lambda` x`:` $sort$ `.` e	lambda expression
		|	e `||` e	or
		|	e `&&` e	and
		|	e `==` e	equals
		|	e `!=` e	not equals
		|	e `<` e	less than
		|	e `>` e	greater than
		|	e `+` e	add
		|	e `-` e	subtract
		|	e `in` e	list containment
		|	e `|>` e	prepend list element
		|	e `++` e	concatenate lists
SortExpr	$sort$	::=	`Bool`	booleans
		|	`Nat`	natural numbers
		|	`Pos`	positive integers
		|	`Int`	integers
		|	`List(`$sort$`)`	lists
		|	x	custom types
		|	$sort$ `->` $sort$	function types
		|	$sort$ `#` $sort$	product of sorts
		|	`struct` $constr$ `|` ... `|` $constr$	structs
ConstrExpr	$constr$	::=	x`(`x`:` $sort$`,...,`x`:` $sort$`)`	constructor
		|	x`(`x`:` $sort$`,...,`x`:` $sort$`)` `?` x	constructor

Figure 2.3. mCRL2 syntax of data expressions.

```
1 [0,1,2] ++ [3,4,5] == [0,1,2,3,4,5]
```

There are also two operators on lists that are not defined in the syntax above: `head` and `tail`. `head` returns the first element of a list and `tail` returns all but the first element of a list. We show an example of applying head and tail to a list.

Example 2.6. Using head and tail.

```
1 head([0,1,2]) == 0
2 tail([0,1,2]) == [1,2]
```

If we apply `head` or `tail` to an empty list, the rewriter gets stuck.

```
1 head([]) == head([])
2 tail([]) == tail([])
```

Lambda expressions define a mapping from inputs to outputs. These input/output mappings can be updated using the update operator (e`[`e`->`e`]`). The output corresponding to

some input can be looked up using $e(e)$. In mCRL2, lambda expressions can be considered functions that can be passed around and updated on the fly.

Example 2.7. Using lambda expressions. We define a function that maps all integers to 0.

```
lambda x: Int . 0
```

We can update this function such that 1 maps to 42.

```
(lambda x: Int . 0)[1->42]
```

When we apply the resulting function with 1 as the argument we get 42.

```
((lambda x: Int . 0)[1->42])(1) == 42
```

All other number still map to 0, e.g.

```
((lambda x: Int . 0)[1->42])(3) == 0
```

It is possible to define new data types using the `sort` keyword. The proper syntax for defining new sorts is shown in Figure 2.4.

SortSpec $\quad sort\_spec \quad ::= \quad$ `sort` x `=` $sort$`;`

Figure 2.4. mCRL2 syntax for defining new sorts.

An inductive data structures can be created using the `struct` keyword. We show an example of defining a custom list datatype.

Example 2.8. Definition of a custom list type.

```
sort ConsList = struct Nil | Cons(value: Int, tl: CustomList)
```

In the next subsection we will see that we can pattern match on `ConsList` expressions. If we want a way to distinguish between `Nil` and `Cons` without pattern matching we can change our definition as follows:

```
sort ConsList = struct Nil ? is_nil
                     | Cons(value: Int, tl: CustomList) ? is_cons
```

`is_nil` and `is_cons` are functions of type `ConsList -> Bool`. `is_nil(x)` is `true` if `x` is `Nil`, or `false` otherwise. `is_cons(x)` is `true` if `x` is a `Cons` constructor, or `false` otherwise. The names of the constructor arguments can be used to get their values:

```
value(Cons(42, Nil)) == 42
tl(Cons(42,Nil)) == Nil
tl(Cons(42,Cons(0,Nil))) == Cons(0,Nil)
value(tl(Cons(42,Cons(0,Nil)))) == 0
```

In the previous example we used the syntax `sort` x `=` $sort$ to introduce a new sort named x. We have not properly defined this syntax yet, but now would be a good time to do so.

2.4.3 Rewrite rules

Rewrite rules are an important part of mCRL2 specifications. They fulfill the role that is often served by function definitions in other programming languages. Rewrite rules tell the rewriter how to rewrite expressions into other expressions, which are usually simpler. The rewriter is a part of mCRL2 that, during model checking or LTS generation, applies rewrite rules to expressions until they reach their simplest form. The following is a rewrite rule that, when applied, rewrites an integer into its successor.

Example 2.9. Example of a simple rewrite rule in mCRL2.

```
map successor: Int -> Int;
var n: Int;
eqn successor(n) = n + 1;
```

A couple of things are happening in this example. First we declare the name of a successor function of sort `Int -> Int`. Then we declare a single rewrite rule with one integer variable `n`. The `var` syntax serves the purpose normally served by function parameters to declare free variables.

The syntax for mappings and rewrite rules is formally defined in Figure 2.5.

MapSpec	*map_spec*	::=	`map` x `:` $sort$`;`	
EqnSpec	*eqn_spec*	::=	var_decl^* eqn_decl^+	
VarDecl	*var_decl*	::=	`var` x`,...,`x `:` $sort$`;`	variable declaration
EqnDecl	*eqn_decl*	::=	`eqn` e `=` e`;`	equation
		|	`eqn` e `->` e `=` e`;`	equation with condition

Figure 2.5. mCRL2 syntax for defining new mappings and rewrite rules.

We can use pattern matching on the left-hand side of a rewrite equation, to deconstruct an inductive structure. This works similarly to pattern matching in functional programming languages. Consider the following example.

Example 2.10. Destructuring the custom inductive type `ConsList`, from Example 2.8, using rewrite rules.

```
map sum: ConsList -> Int;
var n: Int;
    cl: ConsList;
eqn sum(Nil) = 0;
    sum(Cons(n, cl)) = n + sum(cl);
```

Finally, it is possible to provide a condition that needs to hold for a rewrite rule to apply. Variables that are captured on the left-hand side of a rewrite equation can occur in the condition expression. The following example shows how to use conditions to implement a function that computes the Fibonacci sequence.

Example 2.11. Using conditions on rewrite rules.

```
map fib: Int -> Int;
var n: Int;
eqn n <= 1 -> fib(n) = n;
    n > 1 -> fib(n) = fib(n-1) + fib(n-2)
```

The two rewrite rules are preceded by e `->` syntax, where e is a boolean expression.

2.4.4 Actions

Actions are the main building block of mCRL2 processes and represent their observable behavior. They are the same actions that LTSs are made out of. There is one major difference between mCRL2 actions and the actions we have seen so far: mCRL2 actions can be parameterized by data. Figure 2.6 formally defines syntax for specifying actions.

The following examples show how to declare actions with and without data parameters.

Example 2.12. Actions without data parameters.

ActSpec	*act_spec*	::=	`act` x`,...,`x`;`	action
		|	`act` x`,...,`x `:` $sort$`;`	action with data

Figure 2.6. mCRL2 action specification syntax.

```
act Raise, Lower;
```

Example 2.13. Actions with data parameters. Both the buy and the sell action have two parameters of types: `Product` and `Int`. `Product` is a custom sort.

```
sort Product = struct Chocolate | Milk;

act Buy, Sell: Product # Int;
act Discard: Product;
```

By using data parameters we essentially declare a whole set of actions that range over the elements of their parameter sets. For example, the `Discard` action can be seen as a shortcut for the actions `DiscardChocolate` and `DiscardMilk`. Using parameters, however, makes for much more concise notation. Especially if the parameters have many possible values. Note that there are infinitely many `Buy` and `Sell` actions, because it has an integer parameter and the set of integers is infinitely large.

2.4.5 Processes

The final part of an mCRL2 specification is where everything comes together. Process specifications are used to define processes, they are composed of actions and sub-processes. Actions and sub-processes can be composed in two ways: sequentially or using a choice operator. In Figure 2.7 we define mCRL2 process syntax formally.

ProcSpec	*proc_spec*	::=	`proc` x `=` *proc_expr*`;`	process
		|	`proc` x`(`x`:`$sort$`,...,`x`:`$sort$`)` `=` *proc*`;`	process with data
ProcExpr	*proc*	::=	*act*	action
		|	`delta`	deadlock process
		|	`tau`	transparent action
		|	x	process call
		|	x`(`e`,...,`e`)`	process call with data
		|	*proc* `.` *proc*	sequence
		|	*proc* `+` *proc*	choice
		|	`sum` x`:`$sort$`,...,`x`:`$sort$ `.` *proc*	choice over data
		|	`(`e`)` `->` *proc*	guard
		|	`(`e`)` `->` *proc* `<>` *proc*	guard with else
Action	*act*	::=	x	action
		|	x`(`e`,...,`e`)`	action with data

Figure 2.7. mCRL2 process syntax.

We show an example of a process that consists of two actions in sequence.

Example 2.14. A process `P`, that consists of two actions `a` and `b` in sequence. The process implicitly ends in deadlock, because after action `b` there are no more actions to be taken.

```
act a, b;
proc P = a . b;
```

We use the choice operator if we want to specify that, at a certain point in a process, multiple actions can be taken. Consider the following example.

Example 2.15. A process `Q` where, after action `a`, it is possible to either take action `b` or action `c`.

```
act a, b, c;
proc Q = a . (b + c);
```

We can also call sub-processes within process expressions. An example use case of this is to represent a process that repeats infinitely.

Example 2.16. A process `R` that possibly recurses infinitely if the `b` action is taken. Note that the process terminates implicitly after taking the `c` action.

```
act a, b, c;
proc R = a . ((b . R) + c);
```

It is good practice to use the special `delta` process to indicate termination. The `delta` process is sometimes the *deadlock process*, as it represents a state where no actions are possible. Using `delta` instead of terminating implicitly helps mCRL2 with linearization. The next example shows how we improve the processes from previous examples by explicating deadlocks.

Example 2.17. Improved versions of processes from Examples 2.14 to 2.16, by making deadlocks explicit.

```
act a, b, c;
proc P = a . b . delta;
proc Q = a . (b + c) . delta;
proc R = a . ((b . R) + (c . delta));
```

If we want to express choice over all values of a datatype we can use the `sum` operator. Consider the following example over natural numbers.

Example 2.18. Choice over natural numbers. We specify a process where in the initial state there is an action `a(n)` for all natural numbers `n`.

```
act a: Nat;
proc P = sum n: Nat . a(n) . delta;
init P;
```

The above process provides choice between an infinite amount of actions, since the set of natural numbers is infinitely large. This has some interesting implications. First, it is not possible to generate the full state space (i.e. LTS) of this process in finite time. Since the full state space would be infinitely large. It also means that not all μ-formulas can be model checked on this process. Interestingly, for some μ-formulas it remains possible to model check them. For example, $\langle \texttt{a(0)} \rangle \top$ can be verified, because it does not require generation of the full state space. However, a formula such as $\forall n : \texttt{Nat}. \langle \texttt{a}(n) \rangle \top$ cannot be verified in finite time. We have not introduced universal quantification ($\forall$) over data as an operator in μ-calculus yet. This will be formally introduced in Section 2.5. Informally the meaning of the previous formula is "there is an action `a(`n`)` for all natural numbers n."

Often, it makes sense to limit the values a process can choose from, to make sure the state space of the process is finite. This can be done using the guard operator. It has two varieties, one with, and one without an alternative option (i.e. 'else branch'). Consider the following two examples.

Example 2.19. Using a guard operator to limit the state space of a process.

```
act a: Nat;
proc P = sum n: Nat . (n < 10) -> a(n) . delta;
init P;
```

Example 2.20. Using a guard operator with an alternative. The process represents a person who is able to do two things: earn money and buy stuff. The guard operator is used to check if the person has enough money to buy a certain product. If they do not have enough money for a product, the else branch of the guard presents the option to earn money.

```
sort Product = struct Computer | Food;

map price: Product -> Nat;
eqn price(Computer) = 1000;
    price(Food) = 5;

act EarnMoney: Int;
act BuyProduct: Product;

proc P(money: Nat) =
  sum p: Product . (money >= price(p))
    -> BuyProduct(p) . P(abs(money - price(p)))
    <> EarnMoney(100) . P(money + 100);

init P(0);
```

To find out if it is possible to buy a computer at some point, we could use the following μ-formula.

$$\nu X.\langle \texttt{BuyProduct(Computer)} \rangle \top \vee \langle \top \rangle X$$

2.5 Modal μ-calculus with data

In the previous section we have seen how to specify processes with data. We can define custom (inductive) data types, rewrite rules and actions with data. To be able to state properties about these processes, the μ-calculus as previously defined is not sufficient. mCRL2 extends the μ-calculus with constructs to reason about data. In this section we define an extension of the μ-calculus from Section 2.3 with data quantifiers and a data evaluation construct. The result is a logic sometimes referred to as first-order modal μ-calculus.

The constructs we add are universal quantification, existential quantification and an evaluation operator that evaluates arbitrary boolean mCRL2 data expressions to a boolean constant ($\top$ or $\bot$). The extended syntax is defined in Figure 2.8.

μ-formula	ϕ	$::=$	$\ldots$	syntax from Figure 2.1
		$\mid$	$\text{val}(e)$	data expression evaluation
		$\mid$	$\forall x \in sort.\phi$	universal quantification
		$\mid$	$\exists x \in sort.\phi$	existential quantification

Figure 2.8. Extension of μ-calculus syntax with data operators.

In Figure 2.8, x, $sort$ and e are respectively identifiers, sorts and data expressions from mCRL2 specification syntax. The semantics of μ-calculus formulas with data operators are defined as follows.

Definition 2.8. Given an arbitrary LTS $\mathcal{M} = (S, Act, \{\xrightarrow{a}\}_{a \in \text{Act}}, s_0)$, a fixed point variable valuation ρ that maps variables to sets of states, and a valuation σ that maps quantified variables to values, the semantics of μ-calculus formulas are defined by the equations below. The notation $[\![\phi]\!]^{\rho,\sigma}$ means "the states where formula ϕ holds, given valuations ρ and σ." We consider formula ϕ valid for an LTS if it is valid in its initial state, i.e. $s_0 \in [\![\phi]\!]^{\rho,\sigma}$. We use $V(sort)$ to denote the set of values of belonging to $sort$. The notation $[\![e]\!]^{\sigma}$ is used for the evaluated value of an mCRL2 data expression e under valuation σ. Finally, $\phi[d/x]$ is used to denote the formula ϕ where all occurrences of x are substituted by d.

$$
\begin{aligned}
[\![\top]\!]^{\rho,\sigma} &= S \\
[\![\bot]\!]^{\rho,\sigma} &= \varnothing \\
[\![\neg\phi]\!]^{\rho,\sigma} &= S \backslash [\![\phi]\!]^{\rho,\sigma} \\
[\![\phi_1 \wedge \phi_2]\!]^{\rho,\sigma} &= [\![\phi_1]\!]^{\rho,\sigma} \cap [\![\phi_2]\!]^{\rho,\sigma} \\
[\![\phi_1 \vee \phi_2]\!]^{\rho,\sigma} &= [\![\phi_1]\!]^{\rho,\sigma} \cup [\![\phi_2]\!]^{\rho,\sigma} \\
[\![\langle a \rangle \phi]\!]^{\rho,\sigma} &= \{s \in S \mid \exists s' \in S \text{ s.t. } s \xrightarrow{a} s' \text{ and } s' \in [\![\phi]\!]^{\rho,\sigma}\} \\
[\![[a]\phi]\!]^{\rho,\sigma} &= \{s \in S \mid \forall s' \in S \text{ s.t. } s \xrightarrow{a} s' \text{ implies } s' \in [\![\phi]\!]^{\rho,\sigma}\} \\
[\![X]\!]^{\rho,\sigma} &= \rho(X) \\
[\![\mu X.\phi]\!]^{\rho,\sigma} &= \bigcap_{R \subseteq S} [\![\phi]\!]^{\rho[X \mapsto R],\sigma} \\
[\![\nu X.\phi]\!]^{\rho,\sigma} &= \bigcup_{R \subseteq S} [\![\phi]\!]^{\rho[X \mapsto R],\sigma} \\
[\![\forall x \in sort.\phi]\!] &= \bigcap_{d \in V(sort)} [\![\phi[d/x]]\!]^{\rho,\sigma[x \mapsto d]} \\
[\![\exists x \in sort.\phi]\!] &= \bigcup_{d \in V(sort)} [\![\phi[d/x]]\!]^{\rho,\sigma[x \mapsto d]} \\
[\![\text{val}(e)]\!] &= \begin{cases} S & \text{if } [\![e]\!]^{\sigma} = \texttt{true}, \\ \varnothing & \text{if } [\![e]\!]^{\sigma} = \texttt{false}. \end{cases}
\end{aligned}
$$

In the above definition, the LTS and available sorts are defined by an mCRL2 specification. We do not define how to get from an mCRL2 specification to a LTS, as this translation lies outside the scope of this thesis. Additionally, the availability of a relation that evaluates data expressions ($[\![e]\!]^{\sigma}$) is assumed. Refer to Chapter 15 of (Groote and Mousavi 2014) for a complete definition of this relation.

An example of a formula with a quantification over data defined in Example 2.13 is the following.

$$\forall x \in \texttt{Product}.\langle \texttt{Buy}(x) \rangle \top$$

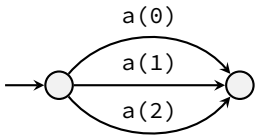
This formula asserts the existence of a `Buy` action for every type of product. A similar formula can assert the existence of a `Buy` action for *some* product, using existential quantification:

$$\exists x \in \texttt{Product}.\langle \texttt{Buy}(x) \rangle \top$$

For an example usage of the data expression evaluation operator, consider the following mCRL2 specification.

```
act a: Int;
proc P = sum i: Int. (i >= 0 && i <= 2) -> a(i) . delta;
init P;
```

This specification results in the following LTS.



We can use the evaluation operator to check for the existence of an action $\texttt{a}(i)$ where $i > 0$, as follows:

$$\exists i \in \texttt{Int}.\langle \texttt{a}(i) \rangle \text{val}(i \texttt{ > } \texttt{0})$$

Chapter 3

Overview of the pattern language: Dyno

In this chapter we introduce the Dyno pattern language. The aim is to build an intuition of how patterns work and what we can do with them. We do this by defining Dyno syntactically and giving characteristic examples of patterns. Before we can discuss Dyno, however, we need to establish an object language that can be pattern matched against.

We establish a programming language that we use as the object of pattern matching in Section 3.1. The language, which we call Imp, is designed to exhibit typical features of an imperative programming language, such as if statements, loops and modifiable state. Then, in Section 3.2 we introduce Dyno's syntax and discuss example patterns. Finally, in Section 5.3 we discuss our Spoofax implementation of translations of Dyno patterns to μ-calculus formulas and Imp programs to mCRL2 specifications.

3.1 Defining an object language: Imp

Before designing the pattern language, we need to have a programming language that we can pattern match on. We call this language the *object language*. To avoid the complexity of dealing with a full-fledged programming language, we instead define a simple toy language. While it should be simple, we want it to have characteristic features of real world programming languages. As such, the language will have constants, variables, unary and binary operators, a branching construct, a looping construct and functions. It is designed to be reminiscent of imperative-style languages like C and hence name it Imp. Programs written in the Imp language have a global state consisting of global variable assignments. When global state is used, an Imp program can be thought of as a class instance where the global variables model class member variables. The syntax of the language is defined in Figure 3.1.

A simplification we make is to not have functions as first-class citizens, i.e. they cannot be passed around as values. Functions can only be referred to directly, using their name. Furthermore, function calls are statements instead of expressions. This simplifies the design of the mCRL2 specification in Section 4.1 somewhat. We introduce an additional *call assign* statement to make sure the return value of a function call can be captured in a variable. Note that we require functions to be annotated with a return type and also require their parameters to be annotated with types. While we do not concern ourselves with defining type checking rules, these annotations later help us infer bounds to limit a program's state space while pattern matching.

Since the semantics of Imp are quite standard and not of special concern, we do not formally define them here. Instead, we rely on an indirect definition of semantics in our model extraction method discussed in Chapter 4.

Type	t	::=	`bool` | `int` | `void`	
Value	v	::=	b	boolean
		|	n	integer
		|	`void`	void
Expr	e	::=	v	constant
		|	x	variable
		|	`global` x	global variable
		|	`-`e	negation
		|	`!`e	not
		|	e `+` e	addition
		|	e `-` e	subtraction
		|	e `&&` e	and
		|	e `||` e	or
		|	e `==` e	equals
		|	e `!=` e	not equals
		|	e `<` e	less than
		|	e `>` e	greater than
Stmt	s	::=	e	expression
		|	x `=` e	assignment
		|	`global` x `=` e	global assignment
		|	`if` (e) s `else` s	if statement
		|	`if` (e) s	if statement (empty else)
		|	`while` (e) s	while loop
		|	x`(`$[e,]^*$`)`	function call
		|	x `=` x`(`$[e,]^*$`)`	call + assignment
		|	`{` $[s;]^*$ `}`	block
		|	`return` e	return
		|	`return`	return void
FunDef	f	::=	`fn` x`(`$[x$ `:` $t,]^*$`)` `->` t s	
ExtFunDef	ef	::=	`fn` x`(`$[\_,]^*$`)` `->` t	
Decl	$decl$	::=	`private` f^*	private functions
		|	`public` f^*	public functions
		|	`external` ef^*	external functions
		|	`global` $[x$ `=` $e]^*$	global assignments
Program	p	::=	$decl^*$	program

Figure 3.1. Syntax of the Imp programming language. An Imp program (p) consists of a list of function and global variable declarations ($decl$). Functions can be *public*, *private* or *external*. Function bodies are composed of statements (s) which are in turn composed of expressions e. Expressions can be of integer or boolean type. We use x^* or $[x]^*$ to denote zero or more occurrences of x and $[e,]^*$ to denote zero or more occurrences of e, separated by commas. Similarly, $x?$ denotes zero or one occurrence of x.

Some Imp syntax is desugared into a core subset of Imp. Refer to Figure 3.2 for a complete overview of desugaring rules. While we only consider the core subset of Imp in the remaining chapters, pattern matching can be applied to the full language by applying the defined desugaring rules when appropriate.

$$[\![\texttt{return}]\!] = \texttt{return void}$$
$$[\![\texttt{if (}e\texttt{) s}]\!] = \texttt{if (}e\texttt{) }s\texttt{ else \{\}}$$
$$[\![x\texttt{(}[e\texttt{,}]^*\texttt{)}]\!] = \texttt{dummy = }x\texttt{(}[e\texttt{,}]^*\texttt{)}$$

Figure 3.2. Desugaring rules for Imp syntax. Return values of function calls are assigned to a reserved identifier called `dummy`.

3.1.1 Examples of Imp programs

To get a feeling for what an Imp program looks like, we give some examples of Imp programs. The following program showcases most of Imp's features. In particular, it has two public functions, a private function and global state variables.

Example 3.1. A program representing a vending machine that initially holds 4 chocolates and 0 coins. Extra coins can be inserted up to the amount of chocolates and chocolates can be ejected, provided there are enough chocolates and coins in the machine.

```
1  global
2    coins = 0
3    chocolates = 4
4
5  public
6    fn insert(amount: int) -> int {
7      max = global chocolates - global coins;
8      diff = min(amount,max);
9      global coins = global coins + diff;
10     return diff
11   }
12
13   fn eject() -> bool {
14     if (global chocolates > 0 && global coins > 0) {
15       global chocolates = global chocolates - 1;
16       global coins = global coins - 1;
17       return true
18     };
19     return false
20   }
21
22 private
23   fn min(a: int, b: int) -> int
24     if (a < b) return a
25     else return b
```

The following is an example of a program with an external function. External functions are *abstract*, in the sense that we do not know their implementation.

Example 3.2. A program with a `main` function that calls an external `print` function.

```
1  public
2    fn main() -> void
3      print(42)
4
```

```
5 external
6   fn print(_) -> void
```

3.2 Introducing Dyno

In this section we introduce the pattern language Dyno. The pattern language is designed to facilitate pattern matching on the models we extract from Imp programs using the mCRL2 specification built in the previous section. We strive for a pattern language that is as intuitive as possible, and therefore support the use of concrete object language syntax inside patterns. Figure 3.3 shows the syntax of Dyno.

Identifier	id	::=	x	regular variable
		|	`@`x	metavariable
MetaType	τ	::=	`statement`	
		|	`expression`	
		|	`identifier`	
		|	`value`	
MetaVarDecl	mv	::=	`var @`x `:` τ	metavariable declaration
Pattern	p	::=	`{{` s `}}`	concrete syntax
		|	`[...]`	ellipsis ('forall')
		|	`<...>`	ellipsis ('exists')
		|	`assert` e	assertion
		|	id`(`[v`,`]`*)`	function call
		|	id`(`[v`,`]`*) ->` v	function call capture
		|	`!`p	negation
		|	p p	sequential composition
Dyno	$dyno$	::=	mv`*` p	Dyno pattern

Figure 3.3. Definition of Dyno pattern syntax.

A Dyno pattern consists of a series of operators that describe a sequential pattern. The operators have the following informal meanings.

- The *concrete syntax* operator (`{{ `s` }}`) is used to match exact syntax of the underlying Imp program.
- *Ellipses* operators (`[...]` and `<...>`) are used to match arbitrary sequences of a program. There are two variants, the *forall* variant (`[...]`) is used if *all* subsequent runtime need to satisfy the remaining pattern, whereas the *exists* variant (`<...>`) is used if *at least one* subsequent runtime path has to satisfy the remaining pattern. If only a single runtime path exists the two are equivalent. Note that we sometimes call the exists and forall ellipses *diamond* and *box* ellipses, respectively.
- The *assertion* operator (`assert` e) can be used to assert that some proposition holds at a certain point in the program. The assertion operator is especially useful to assert things about captured runtime values.
- *Function call* operators are used to match function calls. While function calls can also be matched syntactically using the concrete syntax operator, a dedicated function call operator is needed to match function calls that do not occur syntactically in a program. An example of this is calls to entry point functions. There are two variants, one that captures the returned value (id`(`[v`,`]`*) ->` v) and one that does not (id`(`[v`,`]`*)`).

- *Negation* is used to express that the subsequent pattern should *not* occur.

In addition to the aforementioned operators, Dyno offers metavariables as a means to capture arbitrary statements, expressions, identifiers and runtime values. Metavariables are declared at the top of a Dyno pattern. To support metavariables we augment the syntax of Imp. In Figure 3.4 we show how Imp syntax is extended with Dyno metavariables, as well as capture constructs. The capture operators can be used to capture the runtime values of expressions and the return values of function calls.

Value	v	::=	...	Figure 3.1 definitions
		|	`@`x	metavariable
Expr	e	::=	...	Figure 3.1 definitions
		|	`global` id	global variable
		|	e `->` v	value capture
Stmt	s	::=	...	Figure 3.1 definitions
		|	id `=` e	assignment
		|	`global` id `=` e	global assignment
		|	id`(`[e`,`]`*)`	call
		|	id `=` id`(`[e`,`]`*)`	call + assign
		|	id`(`[e`,`]`*)` `->` v	call value capture
		|	id `=` id`(`[e`,`]`*)` `->` v	call + assign value capture

Figure 3.4. Updated Imp syntax with Dyno metavariables and constructs for capturing runtime values.

3.2.1 Examples of Dyno patterns

We will now show simple examples of Dyno patterns that show how each of the operators work, and how the operators can be combined sequentially.

Example 3.3. A pattern that matches concrete syntax sequentially.

```
1 {{ x = 0 }}
2 {{ print(x) }}
```

The above example expresses that an assignment of `0` to `x` should be immediately followed by a `print(x)` statement. A program where this pattern might be expected to hold is the following.

```
1 public
2   fn main() -> void {
3     x = 0;
4     print(x)
5   }
6
7 external
8   fn print(_) -> void
```

However, as the pattern is currently written it does not match the above program. In fact, it cannot match *any* program, because a program never starts with an assignment statement, but with a call to an entry point function. Therefore, Dyno patterns typically start with an ellipsis operator. For the above example, this results in the following pattern.

```
<...>
{{ x = 0 }}
{{ print(x) }}
```

Using the diamond ellipsis the meaning of the pattern becomes "there is a runtime path to `x = 0`, directly followed by a `print(x)` call." This pattern correctly matches the above program.

As an example of how Dyno differs from purely syntactic pattern matching, consider the following program.

```
public
  fn main() -> void
    if (false) {
      x = 0;
      print(x)
    }

external
  fn print(_) -> void
```

Now, the question is whether our previous pattern matches the above program. If we were to pattern match purely syntactically, a match should indeed be found. However, Dyno takes into account the runtime semantics of a program, thereby figuring out that the code in the if statement can never be reached. So our Dyno pattern from above does not match this program.

The next example shows a use case of metavariables.

Example 3.4. A pattern with metavariables that matches an arbitrary while loop.

```
var @e: expression
var @s: statement
<...>
{{ while(@e) @s }}
```

This pattern can be used to find occurrences of while loops in a program. If a program contains at least one while loop, the pattern matches.

We can use the call operator to match function calls. Consider the following example.

Example 3.5. A pattern that matches a specific function call that returns a specific value.

```
foo() -> 4
```

The above pattern matches programs where a public function foo() is available, that returns `4` in the initial state.

Note that the call operator differs slightly from the concrete syntax operator. For example, the following pattern, though similar to the example above, has a different meaning.

```
{{ foo() -> 4 }}
```

This pattern looks for the occurrence of a syntactic occurrence of a `foo()` function call that returns `4`, which never occurs in the initial state.

Concretely, the latter pattern fails for the following program, while the former succeeds.

```
public
  fn foo() -> int
    return 4
```

Next, we discuss the negation operator. When a negation operator is added in front of a pattern, it flips its meaning. That is, if a pattern matched a certain program before negating, it no longer matches after negating, and vice versa. More interesting are patterns where a negation occurs in the middle, or at least not at the very front. Consider the following example that detects a variable assignment that remains unused.

Example 3.6. A pattern that uses the negation operator.

```
1 var @x: identifier
2 var @e: expression
3 <...> {{ @x = @e }} !<...> {{ @x }}
```

The following is an example that uses the assertion operator to reason about a captured runtime value. The pattern matches if a `print` call is made with a value greater than `4`.

Example 3.7. A pattern with an assertion operator.

```
1 var @v: value
2 <...>
3 print(@v)
4 assert @v > 4
```

Using the concrete syntax operator and expression value capture we can express roughly the same pattern:

```
1 var @e: expression
2 var @v: value
3 <...>
4 {{ print(@e -> @v) }}
5 assert @v > 4
```

The two formulations only differ in the scenario where the print call is the result of an external program calling the public function `print`.

Lastly, we show two examples to highlight the difference between the 'exists', (or 'diamond') ellipsis and the 'forall' (or 'box ellipsis').

Example 3.8. An example of using diamond ellipsis to find the occurrence of a function call.

```
1   <...>
2   {{ foo() }}
```

Example 3.9. An example of using box ellipsis to find the occurrence of a function call on all paths of computation.

```
1   [...]
2   {{ foo() }}
```

An example of a program that matches the diamond ellipsis, but not the box ellipsis variant is the following.

```
1 public
2   fn main() -> void
3     foo()
4   fn bar() -> void {}
5 private
6   fn foo() -> void {}
```

3.2.2 Limiting input parameter spaces using function bounds

Imp programs do not necessarily have a single entry point. Instead, we assume that any public function is an entry point and can be called by an external program at any time after the program is instantiated. One can think of an Imp program as an instance of a class of an object-oriented program.

Whenever a public function with integer and/or boolean parameters is called, its parameters can take on any combination of values from the value sets of its parameters. When the parameters are all booleans this means the amount of possible combinations is 2^n, where n is the amount of boolean parameters. Thus, the input space grows exponentially with the number of boolean parameters. Moreover, when there is even only a single integer parameter, the set of possible input values is already infinite. While during normal execution this is not a concern, as the input values are always known, and we do not care about the other values a parameter *could* have taken on. However, when we are pattern matching on a program, we do not know what the actual inputs to a program are. To be able to match on execution traces of a program, we then need to consider all possible inputs. Since this is infeasible if the input space is infinite, we need a way to limit the input space if we want to be able to pattern match in finite time.

To this end we extend Imp with bound syntax in Figure 3.5. These bounds should not be considered part of Imp, but rather as a part of Dyno. Bound syntax is provided for bounding the input values of (internal) functions of the form `bound` x `=>` e, where x is the parameter to be bounded and e a boolean expression bounding x. Similar syntax is provided for bounding the return values of external functions: `bound out =>` e, where `out` is the only variable in scope of e and represents the return value.

FunDef	f	::=	`fn` x`(`$[x$ `:` t`,]`*`)` `->` t s ib^*
ExtFunDef	ef	::=	`fn` x`(`$[$_`,]`*`)` `->` t $ob?$
InputBound	ib	::=	`bound` x `=>` e
OutputBound	ob	::=	`bound out =>` e

Figure 3.5. Extending Imp function definitions with bound syntax. The notation $x?$ means zero or one occurrence of x, and x^* means zero or more occurrences of x.

Consider the following Imp program as an example.

Example 3.10. A program with a public function that has bounded input parameters.

```
1  public
2    fn add(x: int, y: int) -> int
3      return x + y
4    bound x => x >= -5 && x <= 5
5    bound y => y >= -5 && y <= 5
```

Without bounds on the input parameters, pattern matching might fail due to the state space of the program being infinite. The downside of this is that we only consider a subset of the state space. An example of a pattern that can be expressed about the following program is the following.

```
1 var @v: value
2 var @v': value
3 var @v'': value
4 [...]
5 add(@v,@v') -> @v''
6 assert @v'' == (@v + @v')
```

This pattern asserts that all calls to the `add` function do indeed return the sum of the arguments.

We can also bound the values returned by external functions, this might be necessary if an external function returns an integer. Consider the following example.

Example 3.11. A program where the return value of an external function is bounded.

```
public
  fn main() -> void {
    x = random();
    print(x)
  }

external
  fn print(_) -> void
  fn random() -> int
  bound out => out >= -10 && out <= 10
```

3.3 Practical examples of Dyno patterns

In the previous section we have seen some initial examples of simple Dyno patterns. In this section we focus on more practical examples of patterns that can be expressed with Dyno. We show examples of checking pre- and post-conditions of function calls Section 3.3.1. Patterns for finding (in)correct resource usage are discussed in Section 3.3.2. We wrap up with Section 3.3.3, which discusses a pattern that detects infinite loops.

3.3.1 Checking pre- and post-conditions of function calls

As the first set of examples we devise patterns for checking pre-conditions and post-conditions of functions. In other words patterns that assert conditions on the input values or output values of function calls. A simple example of a post-condition check is a pattern that asserts a condition on the return value of a function call.

Example 3.12. A pattern that expresses a condition on the return value of a function.

```
  var @v: value
  var @r: value
  <...>
  foo(@v) -> @r
  assert @r >= 0 && @r <= 42
```

In practice, one might be more interested in the negated variant of the above pattern. Such a pattern could be used to check whether *all* calls of a specific function return a value within certain bounds. For example, if we wanted to check whether `foo` never returns a value greater than `10`, in a certain program, we could use the following pattern.

```
var @v: value
var @r: value
!<...>
foo(@v) -> @r
assert @r > 10
```

One might be tempted to implement the above pattern without negation by using a box ellipsis instead, for example:

```
var @v: value
var @r: value
[...]
foo(@v) -> @r
assert @r <= 10
```

This would work if the meaning of the pattern was "all `foo` calls return a value of at most `10`." However, this is *not* the actual meaning of the pattern. A correct reading of its meaning would be "all execution paths starting in the initial state lead to a `foo` call that returns a value of at most `10`." It might be, for example, that on a certain path a `foo` call occurs that returns `8`, while afterwards on the same path a call occurs that returns `11`. This second call would not be considered by the box pattern variant, because it already found a satisfactory function call along the path. If one wants to assert something about *all* function calls, the negated pattern above should be used instead.

Pre-condition patterns are very similar, but instead operate on function arguments. Consider the following example that checks whether `foo` is called with the value `true`.

```
<...>
foo(true)
```

To check whether an argument passed to the `foo` function always lies within a certain range we could use the following pattern.

```
var @v: value
<...>
foo(@v)
assert @v < 10 || @v > 20
```

Note that such a pattern only makes sense for private or external functions. Public functions usually need to be bounded by the user, and if the asserted range completely overlaps with the bounds, the pattern always trivially matches the program, or mismatches otherwise.

3.3.2 Correct use of resources with a limited lifetime

We can devise patterns that check whether a resource is used correctly. Consider a resource that is available for a limited timespan during a program's execution. From a certain point onwards, the resource is available and can be interacted with, until it is destroyed and becomes unavailable. A real-world example of such a resource is pointers to (heap) memory in a language like C. After allocation, a pointer can be used to access the underlying memory. When memory is freed, however, the pointer is invalidated and should no longer be used. Attempting to access memory that has been freed is known as a use-after-free bug. While we do not have pointers in Imp, we can still devise an example that demonstrates how we can express a use-after-free pattern in Dyno to search for occurrences of such a bug. Consider the following program that manages a resource with limited lifespan.

```
public
  fn access() -> void
    use()

  fn destroy() -> void
    free()

external
  fn use() -> void
  fn free() -> void
```

The program provides `access` and `destroy` functions as a public interface to interact with a resource. The external functions `use` and `free` represent the functions that actually interact with the underlying resource. We now want to check whether this interface prevents use-after-free bugs, using the following pattern.

Example 3.13. A pattern that expresses the absence of use-after-free bugs.

```
1  !<...>
2  free()
3  <...>
4  use()
```

However, if we match this pattern against the above program, we get `false` as result. Indeed, it is possible for a `use` to occur after a `free`, because the program does nothing to prevent that from happening. We could solve this by adding a global flag that records whether the resource has been freed and checking the flag before calling `use`. The result is the following program.

```
1  global
2    destroyed = false
3
4  public
5    fn access() -> void
6      if (!global destroyed) use()
7
8    fn destroy() -> void {
9      global destroyed = true;
10     free()
11   }
12
13 external
14   fn use() -> void
15   fn free() -> void
```

The use-after-free pattern now successfully matches, indicating that no use-after-free bugs occur.

Another pattern we might want to express is one that asserts that a resource is always disposed of after allocation. This might be useful for checking if file streams are always properly closed, for example. Consider the following example program.

```
1  public
2    fn main() -> void {
3      file = 0;
4      open(file);
5      a = read(file);
6      b = read(file);
7      c = read(file);
8      print(a+b+c);
9      close(file)
10   }
11
12 external
13   fn open(_) -> void
14   fn read(_) -> int
```

```
15  bound out => out >= 0 && out <= 10
16  fn close(_) -> void
17  fn print(_) -> void
```

The program opens a file, reads some integers and closes it after printing the sum of the integers. Note that we need to bound the output of the `read` function to ensure a finite state space. To assert that files are always closed after opening them we use the following pattern, containing a box ellipsis and a double negation.

Example 3.14. A pattern that checks for the proper disposal of resources.

```
1  var @v: value
2  !<...>
3  open(@v)
4  ![...]
5  close(@v)
```

A variant without the double negation that uses two box ellipses instead would not work. This is explained in Section 3.3.1, where we discuss another pattern where it is tempting to replace a negated diamond ellipsis with a box ellipsis.

3.3.3 Detecting an infinite loop

An interesting pattern that we can express in Dyno is one that looks for infinite while loops. The pattern is as follows.

```
1 var @e: expression
2 var @s: statement
3 <...>
4 {{ while(@e) @s }}
5 assert false
```

To understand why this works, we first consider what happens when the pattern encounters a while loop that terminates. In that case we hit the '`assert false`' line which always fails, resulting in a mismatch. However, when an infinite loop is encountered we never exit the loop and the '`assert false`' part of the pattern is ignored.

Note that this pattern only works if the while loop does not induce an infinite state space. This happens if the loop reaches a stable state after a finite amount of iterations, after which every iteration is the same. An example of a program that matches the above pattern is the following.

```
1 public
2   fn main() -> void
3     while(true) {}
```

A peculiar side effect

The above pattern unveils a peculiar side effect of using a greatest fixed point operator in the compilation of while loop concrete syntax. While this allows us to match infinite loops, as previously shown, it also means that *any* sub-pattern after the '`{{ while(@e) @s }}`' part of a pattern holds if there is an infinite loop in a program.

Consider the following example of a pattern with a while loop.

```
1 var @e: expression
2 var @s: statement
3 <...>
```

```
4 {{ while(@e) @s }}
5 {{ print(x) }}
```

If a program contains an infinite loop, this pattern matches, even if no print statement occurs afterwards. One who writes such a pattern likely does not expect this to be its behavior, because it might match programs that have no `print` call after a while loop. We would like to remark however, that if the above pattern is of interest, then infinite loops themselves might be an unwanted side effect. Therefore, one could always run the infinite loop pattern first to check for infinite loops, before running other patterns with while loops.

Chapter 4

Extracting models from Imp programs

Before pattern matching can be applied to programs, they need to be represented in a form that can be readily analyzed. Because we aim to implement Dyno patterns as μ-calculus formulas, this means programs should be represented in some form that μ-formulas can reason about. In this chapter we achieve this by building an mCRL2 specification that extracts models from Imp programs. In Section 4.1 we show how to represent Imp abstract syntax trees (ASTs) in mCRL2 and follow up with a first attempt at translating Imp programs into LTSs. Learning from observations made in Section 4.1, we devise an improved mCRL2 specification in Section 4.2, based on a model extraction technique pioneered by Spaendonck (2024). Finally, in Section 4.3 we show how the specification can be extended with information about internal computation. We end up with a representation of Imp programs suitable for pattern matching with μ-calculus formulas. In Chapter 5 we show how Dyno patterns can be compiled to such formulas.

4.1 First steps toward state space extraction

In the previous chapter we defined the syntax and semantics of our object language: Imp. Eventually we want to pattern match on Imp programs using μ-calculus. To do this, however, we first need to figure out a way to generate LTSs of programs in a way suitable for pattern matching. More specifically, we want to be able to find patterns in runtime traces of programs using concrete syntax. We will be using mCRL2 to specify processes that turn Imp programs into LTSs. Before we get there, however, we need to be able to represent programs in mCRL2.

4.1.1 Representing Imp programs in mCRL2

To represent Imp programs in mCRL2 we define a set of data types to hold their ASTs. In Listing 4.1 we list mCRL2 code that implements the necessary data types. We use `Val`, `Stmt` and `Expr` to represent values, statements and expressions of Imp programs, respectively. These data types closely match the syntactic definitions from Figure 3.1. Only the syntax that is desugared – see Figure 3.2 – has no mCRL2 counterpart. Unary operators are concisely represented using a single `unop` constructor combined with a dedicated data `UnOp` struct that determines the type of unary operator. We do the same for binary operators with the `binop` constructor and `BinOp` struct.

Some constructors have parameters of type `Id`, a type we have not defined yet. Since mCRL2 has no support for strings, we create a custom data type that contains a constructor for each identifier that occurs in a program. This data type therefore depends on the program. If a program contains a function named `print` and a variable `x`, for example, the `Id` struct would be defined as:

```
1 sort Id = struct print | x;
```

```
sort Val = struct BoolV(get_bool: Bool) ? is_bool
                | IntV(get_int: Int) ? is_int
                | VoidV ? is_void;

sort Stmt = struct assign(get_var: Id, get_expr: Expr) ? is_assign
                 | glob_assign(get_var: Id,
                               get_expr: Expr) ? is_glob_assign
                 | ite(get_cond: Expr,
                       get_then: Stmt,
                       get_else: Stmt) ? is_ite
                 | expr(get_expr: Expr) ? is_expr
                 | return(get_expr: Expr) ? is_return
                 | while(get_cond: Expr, get_body: Stmt) ? is_while
                 | call(get_var: Id,
                        get_fun: Id,
                        get_args: List(Expr)) ? is_call
                 | blk(get_stmts: List(Stmt)) ? is_blk;

sort Expr = struct constant(get_val: Val) ? is_constant
                 | variable(get_id: Id) ? is_variable
                 | glob_variable(get_id: Id) ? is_glob_variable
                 | unop(get_unop: UnOp, get_expr: Expr) ? is_unop
                 | binop(get_binop: BinOp,
                         get_left: Expr,
                         get_right: Expr) ? is_binop;

sort UnOp  = struct Not | Neg;
sort BinOp = struct And | Or | Eq | Neq | Add | Sub | Lt | Lte | Gt | Gte;
```

Listing 4.1. mCRL2 data structures representing the abstract syntax of Imp programs.

To illustrate how these data types can be used to represent the AST of an Imp program, we give an example.

Example 4.1. A program that we want to represent in mCRL2 using the data types from Listing 4.1.

```
{
  x = 0;
  while (x < 3) {
    x = x + 1
  }
}
```

The AST that corresponds to this program is given in the next example.

Example 4.2. An AST that corresponds to the program from Example 4.1.

```
sort Id = struct x;
blk([
  assign(x, constant(0)),
  while(binop(Lt, variable(x), constant(3)),
        blk([
```

```
6           assign(x, binop(Add, variable(x), constant(1)))
7         ]))
8 ])
```

4.1.2 A first attempt at extracting LTSs from programs

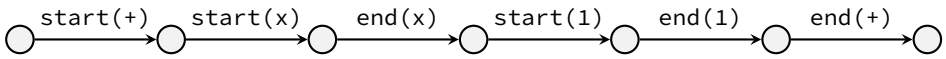
There are multiple ways to approach the problem of transforming programs into LTSs. In this section we explore a method that translates programs to LTSs in a relatively direct manner. The idea is to devise an mCRL2 process that recursively traverses the AST of a program while transforming nodes into sequences of actions. For example, consider the following mCRL2 specification.

Example 4.3. A process that unfolds expressions into a sequence of actions.

```
act start: Expr;
    end: Expr;
proc Unfold(e: Expr) =
    (is_constant(e)) -> start(e) . end(e)
  + (is_variable(e)) -> start(e) . end(e)
  + (is_unop(e))     -> start(e) . Unfold(get_expr(e)) . end(e)
  + (is_binop(e))    -> start(e)
                        . Unfold(get_left(e))
                        . Unfold(get_right(e))
                        . end(e)
```

Using `Unfold` from the above definition we can turn expressions into LTSs. Consider the following example

Example 4.4. Unfolding `binop(Add, variable(x), constant(1))` using `Unfold` results in the following LTS. We abbreviate the binary operator with `+`, the variable with `x` and the constant with `1`.



We can use this LTS to find syntactic patterns in code. Consider the following μ-calculus formula.

$$\langle \texttt{start(+)} \rangle \langle \texttt{start(x)} \rangle \langle \texttt{end(x)} \rangle \langle \texttt{start(1)} \rangle \langle \texttt{end(1)} \rangle \langle \texttt{end(+)} \rangle \top$$

It looks for an expression of the form `x + 1` and matches the LTS from Example 4.4 exactly. What if we want to match an expression at an arbitrary location in a program? We could use a formula with a fixed point operator to achieve this. For example, the following formula looks for the occurrence of a `1` constant somewhere in the LTS.

$$\mu X. \langle \texttt{start(1)} \rangle \langle \texttt{end(1)} \rangle \top \vee \langle \top \rangle X$$

One might wonder why we decided to represent expressions using a `start` and an `end` action, instead of a single `expr` action. The main reason for this is that we need delimiting actions to facilitate metavariables. For example, say we want to look for an arbitrary expression, directly followed by another expression. Such a pattern could be expressed as '`@e 1`', where `@e` represents an arbitrary expression and `1` the constant 'one'. We do not know a priori what `@e` looks like, and whether it has sub-expressions. Because of this we once again resort to the expressive power of the fixed point operator to express the pattern '`@e 1`' using

μ-calculus. We additionally use existential quantification to capture the AST node of the arbitrary expression.

$$\exists e \in \texttt{Expr}.\langle \texttt{start}(e)\rangle \mu X.\langle \texttt{end}(e)\rangle\langle \texttt{start(1)}\rangle\langle \texttt{end(1)}\rangle \top \vee \langle \top \rangle X$$

This formula also matches the above LTS, where e binds to the expression `x`. Remember that we are using abbreviated action labels, so if we want to be precise, then we should say that e binds to `variable(x)`.

While the process `Unfold` provides a simple and intuitive translation from expressions to LTSs, it has a practical problem. It turns out that the `Unfold` process from Example 4.3 is not *regularly linearizable*. The root of the problem is the occurrence of `end` actions *after* the recursive invocations to the `Unfold` process. In the following subsection we briefly go into linearization and regular linearizability. Ultimately we consider this limitation a sign that, while the `Unfold` process is simple and intuitive, it might be the wrong approach. We therefore turn our attention to a different approach in Section 4.2.

Linearization

As a first step towards model checking or LTS generation, process equations have to be linearized. That is, processes are translated into a simpler form known as linear processes. A linear process, P, is a process that consists of a sequence of summands with each a single sum operator, having zero or more variables, one condition, one action and one recursive invocation of the process P. When all processes of an mCRL2 specification are in linear form it is called a linear process specification (LPS). Consider the following example that we borrow from the mCRL2 documentation[1].

Example 4.5. Example of a process `Buffer` and its linear form.

```
proc Buffer = sum m:Nat.read(m).send(m).Buffer;
init Buffer;
```

This process is not linear, as it contains two actions before the recursive invocation of `Buffer`. An example of the same process in linear form is the following.

```
proc Buffer(b: Bool, n: Nat) = sum m: Nat. b -> read(m).Buffer(!b,m)
                             +            !b -> send(n).Buffer(!b,n);
init Buffer(true,0);
```

The mCRL2 toolset offers different ways to linearize a process specification. The default is 'regular' linearization, which leads to linear processes that are simple enough for further analysis or processing. In particular, many optimizations rely on regular linearization. The downside of regular linearization is that it cannot linearize every process that is expressible syntactically. Some processes with actions after (recursive) process invocations, i.e. processes that are not *tail-recursive*, are not regularly linearizable. In particular, our `Unfold` process from above is such a process. mCRL2 offers an alternative mode of linearization, called 'stack' linearization. Using stack linearization, any syntactically correct process can be linearized. However, processes linearized using this method lead to complex linear forms that are hard to process further. As a result, most optimizations cannot be applied to these processes.

[1] mCRL2 documentation on linear process specifications: `https://www.mcrl2.org/web/user_manual/language_reference/lps.html`

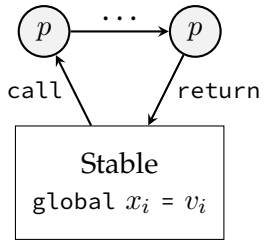


Figure 4.1. Abstract representation of a model extracted from an Imp program.

4.2 Improved state space extraction

In our next approach to LTS extraction from programs we take inspiration from existing research. Spaendonck (2024) presents a technique for extracting behavioral models from C++ programs. They build an mCRL2 specification to achieve this. In this specification, processes are defined that evaluate C++ statement by statement and record the function calls that occur as actions. This essentially results in a LTS that represents the *external* behavior of a program.

In this section, we take inspiration from their approach and build a process specification that is very similar to theirs. The result is a specification that extracts a behavioral model from Imp programs. Figure 4.1 shows an abstract representation of what such an extracted model looks like. We distinguish between two kinds of states: *stable* states and *processing* states. Each stable state represents a different assignment of global variables. From a stable state public functions can be called arbitrarily. Whenever a function is called the system enters a sequence of processing states, where each subsequent function call leads to a new processing state. When the originally called function returns, the system returns to a stable state – possibly with a different assignment to global variables.

In the next section, we modify the specification to also report internal computations between processing states, which is necessary to perform syntax-based pattern matching.

4.2.1 Extracting behavioral models from Imp programs

The behavior of programs is modeled using only two actions: `call` and `ret`. Function calls are modeled using the `call` action. Returning of functions is represented using the `ret` action. Both are defined in Listing 4.2.

```
1 act call: Id # List(Val);
2     ret: Id # Val;
```

Listing 4.2. Definition of a call and return action.

The `call` action has a parameter that holds the function name of the called function, as well as a parameter that contains the list of argument values.

The first process we define is the process that evaluates function calls. This process is defined in Listing 4.3. The process takes as parameters the function name (`pid`), the function body (`body`), the initial function environment (`env`) containing bindings of the function parameters and the global environment (`genv`).

The `P_call` process invokes another process called `P_frame`, which processes the call frame. The definition of this process is shown in Listing 4.4. A call frame (`Frame`) consists of the name of the called function, the remaining statements to be processed, the current local environment and the name of the variable that holds the return value of the called function. The first parameter of `P_frame` specifies the name of the function at the bottom of the call stack – i.e. the function that was called first. The next four parameters (`cid`,`body`,`env`,`ret_var`) specify the stack frame that is currently under evaluation. The last parameter (`stack`) stores the previous

```
sort Env = Id -> Val;

proc P_call(pid: Id, body: Stmt, env: Env, genv: Env) =
  P_frame(pid, pid, [body], env, dummy, genv, []);
```

Listing 4.3. Definition of an mCRL2 process that evaluates a function call. `Env : Id -> Val` is a mapping from identifiers to values.

```
sort Frame =
  struct F(get_pid: Id, get_body: List(Stmt), get_env: Env, get_ret_var: Id);

proc P_frame(pid: Id,
             cid: Id,
             body: List(Stmt),
             env: Env,
             ret_var: Id,
             genv: Env,
             stack: List(Frame)) =
(#body == 0) -> P_return(pid, cid, stack, genv, ret_var, VoidV)
<> (
  is_return(head(body))      -> ...
+ is_expr(head(body))        -> ...
+ is_assign(head(body))      -> ...
+ is_glob_assign(head(body)) -> ...
+ is_blk(head(body))         -> ...
+ is_ite(head(body))         -> ...
+ is_while(head(body))       -> ...
+ (is_call(head(body)) && !is_external(get_fun(head(body))))
                             -> ...
+ (is_call(head(body)) && is_external(get_fun(head(body))))
                             -> ...
);
```

Listing 4.4. Partial definition of an mCRL2 process `P_call` that processes a call frame.

call frames. The `P_frame` process first checks if there is anything left to evaluate, using `#body == 0`. If there are no statements left the function has been fully evaluated without encountering a return statement, so `void` is returned with an invocation of `P_return`. The process `P_return`, which we discuss in more detail shortly, can be found in Listing 4.5. If there are remaining statements to evaluate, the next statement (`head(body)`) is processed. The implementation details of statement evaluation have been elided from Listing 4.4. Next, we fill in the elided details of statement evaluation for each type of statement, starting with the return statement (`return` e).

```
is_return(head(body)) -> P_return(pid,cid,stack,genv,ret_var,
                                  eval(get_expr(head(body)),env,genv))
```

We first process the return statement's expression using a call to the `eval : Expr` × `Env` × `Env -> Val` function. This function takes an expression, a local environment and a global environment and returns a value. It directly implements the semantics for expression implementation. The full implementation of `eval` can be found in Appendix A. The value of the expression is then returned using the `P_return` process. This process is defined in List-

ing 4.5. When it is invoked, it first leaves behind a `ret` action indicating that function `cid` has returned value `ret_val` – i.e. the value of the returned expression. Then, one of two things happens. If the stack is not empty, `P_frame` is invoked to process the next stack frame. If the stack is empty, the function that was called originally has finished execution, and we return to the stable state with the current global environment `Stable(genv)`. The `Stable` process represents the state from which all public functions can be called non-deterministically. We later define it more formally.

```
proc P_return(pid: Id,
              cid: Id,
              stack: List(Frame),
              genv: Env,
              ret_var: Id,
              ret_val: Val) =
    ret(cid, ret_val)
  . (#stack > 0)
      -> P_frame(pid,
                 get_pid(head(stack)),
                 get_body(head(stack)),
                 get_env(head(stack))[ret_var->ret_val],
                 get_ret_var(head(stack)),
                 genv,
                 tail(stack))
      <> Stable(genv);
```

Listing 4.5. Definition of an mCRL2 process that handles returning from a function.

Next, we look at evaluation of the expression statement (e), which simply holds an expression and does nothing with it.

```
is_expr(head(body)) -> tau . P_frame(pid,cid,tail(body),env,ret_var,genv,stack)
```

The expression is simply ignored and an invocation to `P_frame` is made to process the remainder of the function body. Note the use of the `tau` action before the recursive call. The `tau` action is essentially a 'do nothing' action. We need it here to ensure linearization of the process, since the linear form of a process always needs an action before recursive invocations.

Let us have a look at the implementation of the assign statement (x `=` e).

```
is_assign(head(body)) -> tau
                       . P_frame(pid,cid,tail(body),
                                 env[get_var(head(body)) ->
                                   eval(get_expr(head(body)),env,genv)],
                                 ret_var,genv,stack)
```

The inner expression of the assignment is evaluated using `eval` and assigned to x in the local environment `env`. Using the updated local environment `P_frame` evaluates the remainder of the function body.

The implementation for global assignments (`global` x `=` e) is very similar. However, instead of updating the local environment, it updates the global environment `genv`.

```
is_glob_assign(head(body)) -> tau
                            . P_frame(pid,cid,tail(body),env,ret_var,
                                      genv[get_var(head(body)) ->
                                        eval(get_expr(head(body)),env,genv)],
```

```
5                                        stack)
```

A block of statements ({ [s ;]* }) is simply evaluated by unwrapping its list of statements and prepending it to the remainder of the function body. `P_frame` then processes the updated function body.

```
is_blk(head(body)) -> tau . P_frame(pid,cid,
                                    get_stmts(head(body)) ++ tail(body),
                                    env,ret_var,genv,stack)
```

Somewhat more interesting is the evaluation of if statements (`if` (e) s `else` s). First, the branching condition is evaluated. If it evaluates to `true`, we go on to process the `then` branch, otherwise we process the `else` branch.

```
is_ite(head(body)) ->
  (eval(get_cond(head(body)),env,genv) == BoolV(true)) ->
     tau . P_frame(pid,cid,[get_then(head(body))] ++ tail(body),
                   env,ret_var,genv,stack)
  <> tau . P_frame(pid,cid,[get_else(head(body))] ++ tail(body),
                   env,ret_var,genv,stack)
```

Similarly, we implement evaluation of while loops (`while` (e) s).

```
is_while(head(body)) ->
  (eval(get_cond(head(body)),env,genv) == BoolV(true)) ->
     tau . P_frame(pid,cid,[get_body(head(body)), head(body)] ++ tail(body),
                   env,ret_var,genv,stack)
  <> tau . P_frame(pid,cid,tail(body),env,ret_var,genv,stack)
```

If the loop condition evaluates to `true`, the loop body is processed followed by a repetition of the while loop. When the loop condition is not `true` we simply continue evaluating the remainder of the function body.

Implementation of function calls (x `=` x`([`e `,]*`)) is split up into two cases, depending on whether the function is defined as *external* or not. We use `is_external` to determine this. For functions that are not external, i.e. have a concrete implementation within our program, we have the following implementation.

```
(is_call(head(body)) && !is_external(get_fun(head(body))))
  -> call(get_fun(head(body)),evals(get_args(head(body)),env,genv))
   . P_frame(pid,
             get_fun(head(body)),
             [func_body(get_fun(head(body)))],
             make_env(
               func_args(get_fun(head(body))),
               evals(get_args(head(body)),env,genv)),
             get_var(head(body)),
             genv,
             F(cid, tail(body), env, ret_var) |> stack)
```

First we mark the calling of a new function with a `call` action. The call action contains the name of the called function and the values of the call arguments. Instead of calling `P_frame` to process the remainder of the current frame, we build a new frame for the called function. The new frame consists of the name of the called function, its body, a fresh environment, where the values of the argument expressions are mapped to their corresponding identifiers, and the name of the variable to which the return value should be assigned in the calling frame's environment. We use `evals` to evaluate multiple expressions at once and `make_env` to instantiate a new environment with bindings. The `func_body` function maps function names to their

bodies and `func_args` maps function names to a list containing names of their arguments in order. Refer to Appendix A for implementations of `evals` and `make_env`.

We turn our attention to external function calls, of which evaluation is defined follows.

```
(is_call(head(body)) && is_external(get_fun(head(body))))
  -> call(get_fun(head(body)),evals(get_args(head(body)),env,genv))
   . sum v: Val . bound(get_fun(head(body)),dummy,v)
                    -> ret(get_fun(head(body)), v)
                     . P_frame(pid, cid, tail(body),
                               env[get_var(head(body))->v],
                               ret_var, genv, stack)
```

The call action is exactly the same as for regular functions. However, since we do not know what the implementation of the external function is, we cannot know the return value. Instead, we provide non-deterministic choice over all possible values using the `sum` operator. We then use the `bound` function to bound the return value using bounds supplied by the user. Lastly, for each return value a recursive invocation of `P_frame` is made where the return value is assigned to the appropriate variable in the calling function's environment.

That concludes the definition of the `P_frame` process.

4.2.2 Encoding function definitions and bounds

Up until now, all processes we defined are generic in the sense that they are applicable to all Imp programs. Next, we will show how we can instantiate concrete programs by encoding function definitions, global variable assignments and bounds in our mCRL2 specification.

It has already been discussed how we build the `Id` data type in Section 4.1.1, using all identifiers that occur in an Imp program. We do something similar for function definitions, which are defined using four different mCRL2 functions: `func_args`, `func_body`, `is_external` and `bound`.

Consider the following example Imp program.

Example 4.6. A program that we use for example definitions of `func_args`, `func_body`, `is_external` and `bound`.

```
global
  x = 0

public
  fn foo(a: int) -> void
    baz(a)

  fn bar(b: int) -> int {
    global x = b;
    return b
  }
  bound b => b == 10

private
  fn baz(a: int) -> void
    print(a)

external
  fn print(_) -> void
  fn random() -> int
```

```
21     bound out => out >= 0 && out <= 42
```

For this program, the `func_args` function is instantiated as follows.

```
1 map func_args: Id -> List(Id);
2 eqn func_args(foo) = [a];
3     func_args(bar) = [b];
4     func_args(baz) = [a];
```

Note that we do not specify arguments of external functions, as those are ignored. The `func_body` function is defined like this.

```
1 map func_body: Id -> Stmt;
2 eqn func_body(foo) = call(baz, [variable(a)]);
3     func_body(bar) = blk([glob_assign(x, variable(b)),
4                           return(variable(b))]);
5     func_body(baz) = call(print, [variable(a)]);
```

Next, we define the function that indicates which functions are external.

```
1 map is_external: Id -> Bool;
2 eqn is_external(foo) = false;
3     is_external(bar) = false;
4     is_external(baz) = false;
5     is_external(print) = true;
6     is_external(random) = true;
```

In this case, `print` and `random` are external functions. To finalize the specification of function definitions we provide a `bound` function. This function takes the name of a function as first parameter, the name of a bounded argument second and the value to check as the last parameter.

```
1 map bound: Id # Id # Val -> Bool;
2 var v: Val;
3 eqn (is_int(v))  -> bound(foo, a, v) = true;
4     (!is_int(v)) -> bound(foo, a, v) = false;
5     (is_int(v))  -> bound(bar, b, v) =
6       get_bool(eval(binop(Eq,variable(b),constant(IntV(10))),
7                     empty_env[b->v],
8                     empty_env));
9     (!is_int(v)) -> bound(bar, b, v) = false;
10    (is_int(v))  -> bound(baz, a, v) = true;
11    (!is_int(v)) -> bound(baz, a, v) = false;
12    (is_void(v))  -> bound(print, dummy, v) = true;
13    (!is_void(v)) -> bound(print, dummy, v) = false;
14    (is_int(v))  -> bound(random, dummy, v) =
15      get_bool(eval(
16        binop(And,
17              binop(Gte,variable(out),constant(IntV(0))),
18              binop(Lte,variable(out),constant(IntV(42)))),
19        empty_env[out->v],empty_env));
20    (!is_int(v)) -> bound(random, dummy, v) = false;
```

Each internal function has two rules per argument – so a function with 3 arguments would have 6 rules. If the input value does not match the type of the argument, the `bound` function always returns false. If the types do match, we do one of two things. The bound

function always returns `true` if there is no bound associated with the argument. If the types do match, the bound associated with the argument is evaluated using the `eval` function. Note that `empty_env` is an alias for `lambda id: Id . VoidV`, i.e. the function that maps all variables to the `void` value.

For external functions, instead of bounding arguments, the return values are bounded. Since the argument parameter is unused for external functions, we use a placeholder `dummy` value as the second argument to the `bound` function. Otherwise, bounds of external functions work exactly the same as bounds of internal functions.

Before we can instantiate the process we need one more piece of data, which is the initialization of the global environment. In our example, there is only a single global variable `x`, which is initialized to `0`.

```
map glob_init: Env;
eqn glob_init = empty_env[x->eval(constant(IntV(0)),empty_env,empty_env)];
```

Now we define the stable state process as the non-deterministic choice between the stable states of all public functions.

```
proc S(pid: Id, genv: Env) =
  sum vals: List(Val) .
    (is_within_bounds(pid, vals)) ->
      call(pid, vals)
    . P_call(pid,func_body(pid),make_env(func_args(pid),vals),genv);

proc Stable(genv: Env) = S(foo,genv)+S(bar,genv);
```

The `is_within_bounds` function is a helper function that checks the bounds of all arguments of a function at once. The `S` process leaves behind a `call` action and an invocation of `P_call` for all input combinations of the function that are within bounds.

Finally, the program can be instantiated as follows.

```
init Stable(glob_init);
```

4.2.3 Example of extracting a model from a program

To conclude this subsection, we show what the behavioral model of the following program looks like when extracted using the above mCRL2 specification.

Example 4.7. Example program of which we extract a behavioral model. The LTS corresponding to the extracted model of this program is shown in Figure 4.2.

```
  global
    level = 0
  public
    fn raise() -> bool
      if (global level < 2) {
        global level = global level + 1;
        return true
      } else return false
    fn lower() -> bool
      if (global level > 0) {
        global level = global level - 1;
        return true
      } else return false
```

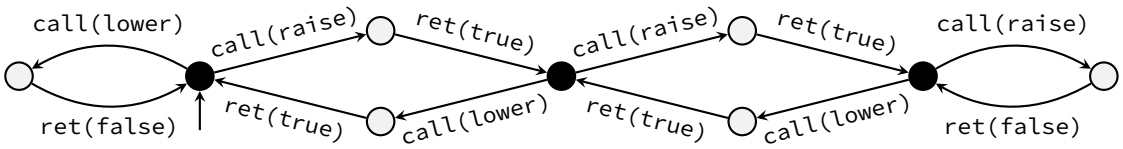


Figure 4.2. Labeled transition system corresponding to the program of Example 4.7. For brevity, argument lists are omitted from `call` actions and function names are omitted from `return` actions. Stable states are colored black, and processing states are colored gray.

4.3 Extending state spaces with internal computation

With the mCRL2 specification devised in the previous section, a behavioral model can be extracted from programs. While it is possible to reason about the *external* behavior of programs, there is not yet a way to reason about the internal computations. In order to pattern match using concrete syntax, on top of function calls, we also need to report the internal computations that happen within functions. In this section the `P_frame` and `P_return` processes defined in Section 4.1 will be modified to include actions that make internal computations visible in the extracted model. Particularly, we add actions that contain the AST information of statements and expressions in order of evaluation.

In Section 4.3.1 we add a new process `P_expr` for unfolding expressions. Section 4.3.2 modifies the mCRL2 specification to support actions for statements in the extracted model. Finally, in Section 4.3.3 we address an issue caused by nested function calls that leads to ambiguities when pattern matching.

4.3.1 Adding support for expression syntax

Whenever a statement is encountered that has sub-expressions, we want to unfold those sub-expressions into actions. We define the following actions for expressions.

```
act start_expr: Expr;
    end_expr: Expr # Val;
```

The `start_expr` action only holds the AST information of an expression, whereas the `end_expr` additionally holds the value that the expression evaluates to.

Now consider the expression statement, which consists of a single sub-expression. The current evaluation rule for expression statements is as follows.

```
is_expr(head(body)) -> tau
                     . P_frame(pid,cid,tail(body),env,ret_var,genv,stack)
```

Assuming we have a process `P_expr` at our disposal, we could simply place an invocation to `P_expr` right before `P_frame`.

```
is_expr(head(body)) -> tau
                     . P_expr(get_expr(head(body)))
                     . P_frame(pid,cid,tail(body),env,ret_var,genv,stack)
```

This, however, leads to a process that is not regularly linearizable, as we have seen in Section 4.1.2. In this case the cause of the problem is the invocation of `P_expr` before `P_frame`. We therefore propose a different formulation, where we *replace* the `P_frame` invocation with a `P_expr` invocation, as follows.

```
is_expr(head(body)) -> tau
                     . P_expr([get_expr(head(body))],
                              pid,cid,tail(body),env,ret_var,genv,stack)
```

Here, in addition to the stack frame parameters that would otherwise be passed to `P_frame`, we pass a stack of expressions as the first argument to `P_expr`. The idea is that `P_expr` first processes the expression stack, and once it is done it hands control back over to the `P_frame` process. In Listing 4.6 we define the scaffolding of the `P_expr` process.

```
proc P_expr(exprs: List(Expr),
            pid: Id, cid: Id, body: List(Stmt), env: Env,
            ret_var: Id, genv: Env, stack: List(Frame)) =
(#exprs == 0) -> P_frame(pid,cid,body,env,ret_var,genv,stack) <>
(
  (is_constant(head(exprs))
   || is_variable(head(exprs))
   || is_glob_variable(head(exprs))) -> ...
+ is_unop(head(exprs))               -> ...
+ is_binop(head(exprs))              -> ...
);
```

Listing 4.6. Partial definition of an mCRL2 process that processes expressions.

The process first checks whether there are any expressions left to process using `#exprs == 0`, if not, control is immediately handed back over to the `P_frame` process. If there is an expression left, we distinguish between three cases. The expression has no sub-expressions, i.e. is a constant or a (global) variable, the expression has a single sub-expression (unary operators) or it has two sub-expressions (binary operators).

The first case is the simplest, and we implement it as follows.

```
(is_constant(head(exprs))
 || is_variable(head(exprs))
 || is_glob_variable(head(exprs)))
   -> start_expr(head(exprs))
    . end_expr(head(exprs),eval(head(exprs),env,genv))
    . P_expr(tail(exprs),pid,cid,body,env,ret_var,genv,stack)
```

The atomic expression is unfolded into a `start_expr` action, directly followed by an `end_expr` action. Note that the `end_expr` action also holds the value that the expression evaluates to. A recursive invocation of `P_expr` processes the remaining expressions on the stack.

Expressions with sub-expressions are more involved. Because we have to make sure the process linearizes, we cannot simply nest a recursive call in-between the `start_expr` and `end_expr` actions, such as:

```
start_expr(_) . P_expr(_) . end_expr(_)
```

Instead, we propose a solution that postpones the `end_expr` action to a new evaluation step. To achieve this we add a new auxiliary constructor to the `Expr` data type.

```
sort Expr = struct ...
                 | end_expr'(get_expr: Expr) ? is_end_expr';
```

The constructor is marked with an apostrophe to emphasize that it is not an actual expression. Using this auxiliary constructor we can define the rule for unary operators as follows.

```
is_unop(head(exprs)) -> start_expr(head(exprs))
                      . P_expr([get_expr(head(exprs)),
                                end_expr'(head(exprs))] ++ tail(exprs),
                                pid,cid,body,env,ret_var,genv,stack)
```

After the `start_expr` action, a recursive invocation of the `P_expr` is made. In this invocation, the inner expression is added to the stack, as well as the auxiliary `end_expr'` constructor.

An additional rule needs to be defined to process this auxiliary constructor. We use the following implementation for this rule, that simply translates the constructor into the `end_expr` action.

```
is_end_expr'(head(exprs)) -> end_expr(get_expr(head(exprs)),
                                      eval(get_expr(head(exprs)),env,genv))
                           . P_expr(tail(exprs),pid,cid,body,env,
                                    ret_var,genv,stack)
```

The rule for binary operators is similar to the rule for unary operators, and is defined as follows.

```
is_binop(head(exprs)) -> start_expr(head(exprs))
                       . P_expr([get_left(head(exprs)),
                                 get_right(head(exprs)),
                                 end_expr'(head(exprs))] ++ tail(exprs),
                                pid,cid,body,env,ret_var,genv,stack)
```

Having defined the `P_expr` process, we continue updating the evaluation rules for statements. The only statement without sub-expressions is the block statement, so we can leave its rule untouched. In all other rules, we substitute the `P_frame` invocation for a `P_expr` invocation with a list of all sub-expressions as the first argument. For assign statements, this leads to the following rule.

```
is_assign(head(body)) -> tau
                       . P_expr([get_expr(head(body))],
                                pid,cid,tail(body),
                                env[get_var(head(body))->
                                    eval(get_expr(head(body)),env,genv)],
                                ret_var,genv,stack)
```

There is a subtle problem with this rule. We pass the wrong environment to the `P_expr` process. The inner expression of the assignment needs to be evaluated in the environment *before* the environment is updated. However, currently the updated environment is passed to the `P_expr` process. A solution to this is to split evaluation of the assign statement into two phases. First process the inner expression of the assignment, then process the assignment itself. We implement this by adding an auxiliary constructor to the `Stmt` data type.

```
sort Stmt = struct ...
                 | assign'(get_var: Id, get_expr: Expr) ? is_assign';
```

The first phase of assignment evaluation now becomes:

```
is_assign(head(body)) -> tau
                       . P_expr([get_expr(head(body))],pid,cid,
                                assign'(get_var(head(prog)),
                                        get_expr(head(prog)))
                                  |> tail(body),
                                env,ret_var,genv,stack)
```

Here we invoke the `P_expr` process and prepend the auxiliary `assign'` constructor to the body to kick off the second phase. The second phase is defined as follows.

```
is_assign'(head(body)) -> tau
                        . P_frame(pid,cid,tail(body),
                                  env[get_var(head(body))->
```

```
4                                       eval(get_expr(head(body)),env,genv)],
5                                   ret_var,genv,stack)
```

Note that the `is_assign'` rule is exactly the same as the old version of the `is_assign` rule. We can apply exactly the same trick to the `is_glob_assign` rule, by adding a `glob_assign'` constructor and splitting evaluation into two phases. For if statements and while loops we can simply replace the `P_frame` call by a `P_expr` call to process their condition expressions.

A few more rules remain to be updated, namely those for return statements and internal and external function calls. We first address function calls. Because we want the actions for the argument expressions of function calls to appear *before* the `call` action, we also need to split up the evaluation rules for function calls into two phases. The first phase processes the argument expressions and is the same for internal and external calls.

```
1 (is_call(head(body))) -> tau
2                         . P_expr(get_args(head(body)),pid,cid,
3                                  [call'(get_var(head(body)),
4                                         get_fun(head(body)),
5                                         get_args(head(body)))] ++ tail(body),
6                                   env,ret_var,genv,stack)
```

This rule processes the argument expressions and replaces the `call` constructor with the auxiliary `call'` constructor to proceed to the second phase. The rules for the second phase (`is_call'`) of external and internal calls are simply copies of their old `is_call` counterparts.

Finally, we address the rules for return statements, making sure its inner expression is properly represented in the LTS. To achieve this, we once again split up its evaluation into two phases. The first phase handles the expression, while the second phase invokes the call to the `P_return` process.

```
1 is_return(head(body)) -> tau
2                         . P_expr([get_expr(head(body))],
3                                  pid,cid,
4                                  [return'(get_expr(head(body)))] ++ tail(body),
5                                  env,ret_var,genv,stack)
6
7 is_return'(head(body)) -> P_return(pid,cid,stack,genv,ret_var,
8                                     eval(get_expr(head(body)),env,genv))
```

4.3.2 Adding support for statement syntax

To support statement syntax, we introduce two new actions: `start_stmt` and `end_stmt`. These actions carry the AST information of the statements that are encountered during the evaluation of function bodies.

```
1 act start_stmt, end_stmt: Stmt;
```

Next, we incorporate these actions into the `P_frame` process, starting with the evaluation rule for expressions statements `is_expr`. For the start action, we can simply replace the `tau` action with a `start_stmt`. The result is as follows.

```
1 is_expr(head(body)) -> start_stmt(head(body))
2                       . P_expr([get_expr(head(body))],
3                                pid,cid,tail(body),env,ret_var,genv,stack)
```

To ensure actions appear in evaluation order, we want the `end_stmt` action to appear *after* the actions of the inner expression. For this we need one last auxiliary statement constructor, similar to the auxiliary `end_expr'` constructor.

```
sort Stmt = struct ...
                  | end_stmt'(get_stmt: Stmt) ? is_end_stmt'
```

Now, an `end_stmt'` can be prepended to the remainder of the body in the invocation of `P_expr`. Resulting in the following rule.

```
is_expr(head(body)) -> start_stmt(head(body))
                     . P_expr([get_expr(head(body))],pid,cid,
                              end_stmt'(head(body)) |> tail(body),
                              env,ret_var,genv,stack)
```

An additional rule is necessary to deal with the auxiliary constructor. We define it as follows.

```
is_end_stmt'(head(body)) -> end_stmt(get_stmt(head(body)))
                          . P_frame(pid,cid,tail(body),env,ret_var,genv,stack)
```

For the assignment statement we can use the two phase evaluation to our advantage. In the first phase, we replace `tau` by a `start_stmt` action and in the second phase, which occurs after processing the expression, we replace `tau` by an `end_stmt`.

```
is_assign(head(body))  -> start_stmt(head(body))
                        . P_expr(...)
is_assign'(head(body)) -> end_stmt(assign(get_var(head(body)),
                                          get_expr(head(body))))
                        . P_frame(...)
```

Note that we make sure to use the `assign` node as argument to the `end_stmt` action, instead of the auxiliary `assign'` node. This saves us some headaches with pattern matching later down the line.

Exactly the same can be done for global assignments:

```
is_glob_assign(head(body))  -> start_stmt(head(body))
                             . P_expr(...)
is_glob_assign'(head(body)) -> end_stmt(glob_assign(get_var(head(body)),
                                                    get_expr(head(body))))
                             . P_frame(...)
```

The rule for block statements is updated by substituting `tau` for a `start_stmt` and adding an auxiliary `end_stmt'` before the remaining body in the recursive invocation of `P_frame`.

```
is_blk(head(body)) -> start_stmt(head(body))
                    . P_frame(pid,cid,
                              get_stmts(head(body))
                                ++ [end_stmt'(head(body))]
                                ++ tail(body),
                              env,ret_var,genv,stack)
```

While perhaps not obvious, this rule for blocks has a problem. The root of the problem lies in the possibility of early termination due to return statements. If somewhere in the block a return statement occurs, the `end_stmt'` constructor is never reached, and we end up with an unmatched `start_stmt` action. This is problematic for pattern matching. The solution we propose is to modify the signatures of `P_frame`, `P_return`, `P_expr` and the frame data type `Frame` to include a list of unclosed statements. We do this by adding the `end_stmts` parameter to the aforementioned processes:

```
P_frame(..., end_stmts: List(Stmt)) = ...
P_return(..., end_stmts: List(Stmt)) = ...
P_expr(..., end_stmts: List(Stmt)) = ...
```

Additionally, we add the following parameter to the `Frame` data type constructor.

```
sort Frame = struct F(..., get_end_stmts: List(Stmt));
```

We update the `P_return` process such that it closes all statements in the `end_stmts` list before returning. The implementation is as follows:

```
proc P_return(..., end_stmts: List(Stmt)) =
(#end_stmts > 0)
  -> end_stmt(head(end_stmts))
   . P_return(..., tail(end_stmts))
  <> ret(cid, ret_val)
   . (#stack > 0)
       -> P_frame(..., get_end_stmts(head(stack)))
       <> Stable(genv);
```

Details that remain unchanged are omitted and abbreviated using '...'.

Whenever the `is_end_stmt'` rule is applied, the first statement has to be removed from the `end_stmts` list:

```
is_end_stmt'(head(body)) -> end_stmt(get_stmt(head(body)))
                          . P_frame(..., tail(end_stmts))
```

The new version of the block statement evaluation rule now becomes:

```
is_blk(head(body)) -> start_stmt(head(body))
                    . P_frame(..., head(body) |> end_stmts)
```

In general, every rule that uses a recursive call where the auxiliary `end_stmt'` constructor appears, also needs to prepend the corresponding statement to the `end_stmts` list.

Of the previously mentioned rules in this section that is only the `is_expr` rule. We modify it as follows.

```
is_expr(head(body)) -> start_stmt(head(body))
                     . P_expr(..., head(body) |> end_stmts)
```

Next, we modify the rule for if statements, such that it has statement actions.

```
is_ite(head(body)) ->
  (eval(get_cond(head(body)),env,genv) == BoolV(true)) ->
      start_stmt(head(body))
    . P_expr(...,
             [get_then(head(body)), end_stmt'(head(body))] ++ tail(body),
             ...,
             head(body) |> end_stmts)
  <>  start_stmt(head(body))
    . P_expr(...,
             [get_else(head(body)), end_stmt'(head(body))] ++ tail(body),
             ...,
             head(body) |> end_stmts)
```

In the implementation above we add a `start_stmt` action and make sure it is eventually properly matched using the `end_stmt'` auxiliary and by prepending the if statement (`head(body)`) to the `end_stmts` list.

The evaluation rule for while loops has to be split up into two phases. This is because we only want `start_stmt` and `end_stmt` actions to appear once per while loop, not for every iteration of the loop. The first phase then takes care of evaluating the loop condition, the first loop iteration and arranging `start_stmt` and `end_stmt` actions. The second phase is then used for subsequent iterations of the for loop. The result is as follows.

```
is_while(head(body)) ->
  (eval(get_cond(head(body)),env,genv) == BoolV(true)) ->
      start_stmt(head(body))
    . P_expr([get_cond(head(body))],
             ...,
             [get_body(head(body)),
              while'(get_cond(head(body)), get_body(head(body))),
              end_stmt'(head(body))] ++ tail(body),
             ...,
             head(body) |> end_stmts)
  <>  start_stmt(head(body))
    . P_expr([get_cond(head(body))],
             ...,
             end_stmt'(head(body)) |> tail(body),
             ...,
             head(body) |> end_stmts)
is_while'(head(body)) ->
  (eval(get_cond(head(body)),env,genv) == BoolV(true)) ->
      tau
    . P_expr([get_cond(head(body))],
             ...,
             [get_body(head(body)), head(body)] ++ tail(body),
             ..., end_stmts)
  <>  tau
    . P_expr([get_cond(head(body))],
             ..., end_stmts)
```

Note that in the second phase there are still `tau` actions. We do not replace them with more concrete actions, they simply remain to ensure linearization is possible. When implementing while loop pattern matching we must remember to account for these `tau` actions.

Next, we augment the call rules to account for statement actions. Once more, we replace a `tau` action by a `start_stmt` action and make sure it is properly closed off using an `end_stmt'` and by prepending to the `end_stmts` list. This is done for the `is_call` rule.

```
(is_call(head(body))) -> start_stmt(head(body))
                       . P_expr(get_args(head(body)),
                                ...,
                                [call'(get_var(head(body)),
                                       get_fun(head(body)),
                                       get_args(head(body))),
                                 end_stmt'(head(body))] ++ tail(body),
                                ...,
                                head(body) |> end_stmts)
```

In the `is_call'` rules, we make sure to properly propagate the `end_stmts` parameter.

```
(is_call'(head(body)) && !is_external(get_fun(head(body))))
  -> ...
   . P_frame(..., F(..., end_stmts) |> stack, end_stmts)

(is_call'(head(body)) && is_external(get_fun(head(body))))
  -> ...
   . P_frame(...,end_stmts)
```

Lastly, we update the rules for return statements to include statement actions and correctly propagate the `end_stmts` list.

```
is_return(head(prog))  -> start_stmt(head(prog))
                        . P_expr(..., head(prog) |> end_stmts)
is_return'(head(prog)) -> P_return(..., end_stmts)
```

4.3.3 Addressing pattern matching issues

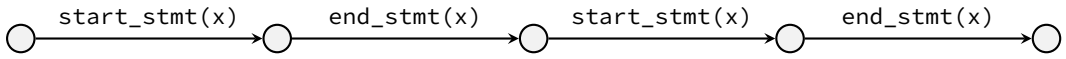
Our mCRL2 specification is almost ready for pattern matching. We are left with one last problem to contend with. Eventually when we model metavariables to match arbitrary statements, we need to find matching `start_stmt` and `end_stmt` actions. We do this with a fixed point, similar to the one seen before in Section 4.1:

$$\exists s \in \texttt{Stmt}.\langle \texttt{start\_stmt}(s)\rangle \mu X.\langle \texttt{end\_stmt}(s)\rangle \top \vee \langle \top \rangle X$$

This formula matches any statement, no matter how many nested statements or subexpressions it has. On issue with this formula is that it is possible to match the wrong `end_stmt` action while still evaluating to true. For example, consider the following Imp snippet.

```
{
  x;
  x
}
```

If we leave out expression actions and abbreviate `expr(variable(x))` as `x`, the LTS that corresponds to the inner part of the block statement is as follows.



The formula above correctly matches the first `start_stmt` action, binding s to the statement `expr(variable(x))`. The problem is that, due to the fixed point and particularly the $\langle \top \rangle X$ part, it is possible to 'step over' the first `end_stmt` and match the second `end_stmt(x)` action instead.

A solution to this problem is to, instead of allowing any action before the fixed point recursion on the right-hand side of the disjunction, we allow all actions *not equal* to the `end_stmt` action that we are looking for. A formula that expresses this is the following.

$$\exists s \in \texttt{Stmt}.\langle \texttt{start\_stmt}(s)\rangle \mu X.\langle \texttt{end\_stmt}(s)\rangle \top \vee \langle \overline{\texttt{end\_stmt}(s)} \rangle X$$

This formula correctly only matches the first `end_stmt(x)`. We can use an analogous formula to match arbitrary expressions.

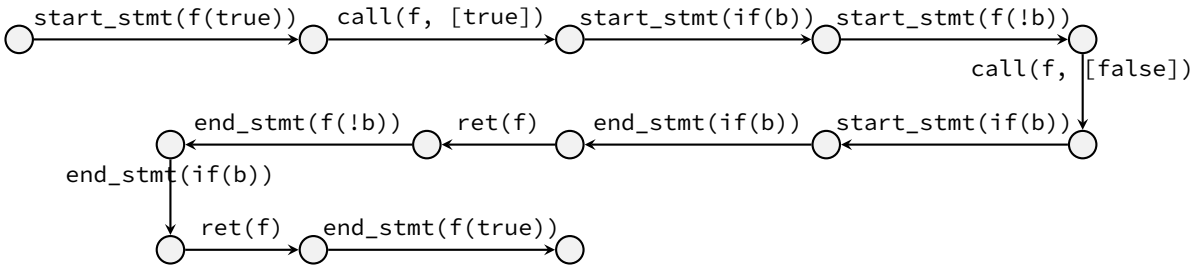
While fixing this issue required no changes to the mCRL2 specification, the next problem does. The issue is similar in nature, in that it involves an ambiguity when searching for a matching `end_stmt`. The ambiguity arises due to the presence of function calls. Consider the following Imp snippet as an example.

```
f(true)
```

With function `f` defined as follows:

```
fn f(b: bool) -> void
  if (b) f(!b)
```

The, simplified, LTS corresponding to this snippet takes the following form.



There are two ambiguities in this LTS. The first one has to do with there being two pairs of `start_stmt(if(b))` and `end_stmt(if(b))` actions. There is no way to distinguish between the two. And in fact, if we use the following modal μ-formula to find an arbitrary if-statement, it will match, but in doing so matches the wrong `end_stmt`.

$$\mu X_0.\phi \vee \langle\top\rangle X_0$$

Where

$$\phi := \exists e \in \texttt{Expr}.\langle \texttt{start\_stmt(if(}e\texttt{))}\rangle \mu X_1.\langle \texttt{end\_stmt(if(}e\texttt{))}\rangle \top \vee \langle \overline{\texttt{end\_stmt(if(}e\texttt{))}}\rangle X_1$$

The μ-formula has been split up into an outer and an inner part, such that it is easier to digest. The outer part models what we call an *ellipsis pattern*, it steps through the LTS until ϕ matches. The inner part (ϕ) is a formula that matches an arbitrary if-statement.

We find that this formula has two matches, but we consider only one of these matches to be correct. The correct match is the *inner* if-statement in the LTS: the second `start_stmt(if(b))`, followed directly by the first `end_stmt(if(b))`. This is the if-statement of the second call to the `f` function. The wrong match is the *outer* if-statement. While it correctly matches the first `start_stmt(if(b))`, the wrong `end_stmt(if(b))` is matched. This is because ϕ always finds the first matching `end_stmt(if(b))`, which in this case is the `end_stmt` corresponding to the *inner* if-statement.

In essence, the root of the problem is the indistinguishability of the `end_stmt(if(b))` actions. Due to this, we find that attempting to change the μ-formula does not lead to an elegant solution. The only way to find the right `end_stmt` is by keeping count of nested `start_stmt` actions.

Instead of changing the formula, we propose to update the mCRL2 specification one last time. To be able to distinguish identical nested statements, we add an integer as identifier to statement actions, as follows.

```
1 act start_stmt, end_stmt: Stmt # Int;
```

Now the question is, what values to assign as identifiers to get rid of the ambiguity described above. We recall that this ambiguity is caused by nested function calls, and therefore propose the identifier to be a counter that increases every time a nested function call is made. Instead of introducing a new parameter to our processes, we realize that an excellent candidate for this counter is the existing `stack` parameter. This parameter keeps track of the stack frames corresponding to nested function calls. It is an excellent candidate to be our counter, because its size increases every time a nested function call is made and decreases every time a function returns.

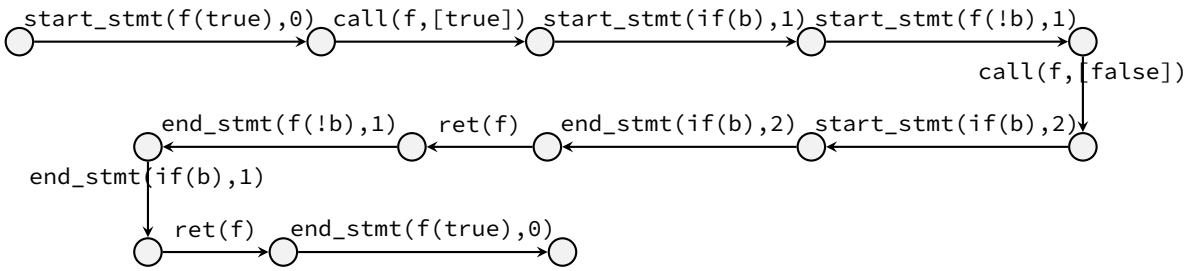
To accommodate the new `start_stmt` and `end_stmt` actions we change every occurrence of `start_stmt` in `P_frame` as follows.

```
start_stmt(_) => start_stmt(_,#stack)
```

Likewise, we change every occurrence of `end_stmt` in `P_frame` and `P_return` as follows.

```
end_stmt(_) => end_stmt(_,#stack)
```

After these changes the LTS of our previous example changes to the following.



Accounting for the new identifiers, we update our formula to find arbitrary if-statements as follows.

$$\mu X_0.(\phi) \vee \langle \top \rangle X_0$$

Where

$$\phi := \exists e \in \texttt{Expr}, i \in \texttt{Int}.\langle \texttt{start\_stmt(if(}e\texttt{),}i\texttt{)} \rangle \mu X_1.\langle \texttt{end\_stmt(if(}e\texttt{),}i\texttt{)} \rangle \top \vee \langle \overline{\texttt{end\_stmt(if(}e\texttt{),}i\texttt{)}} \rangle X_1$$

With that, our mCRL2 specification is almost finished. There is a final ambiguity issue to solve, which is closely related to the one we just fixed. This ambiguity is also related to nested function calls and in fact is also present in the example above. The problem is that we do not know which `ret(f)` action belongs to which `call(f,_)` action. To fix this, we use the same solution we used for disambiguating statement actions. We add an integer identifier to `call` and `ret` actions:

```
1 act call: Id # List(Val) # Int;
2     ret: Id # Val # Int;
```

The `call` action for internal functions in `P_frame` is changed to `call(_,_,#stack)`. The `ret` action in `P_return` is changed to `ret(_,_,#stack-1)`. Here we subtract one from the stack size, because the stack size of the calling function, which is one step up the call-hierarchy, was used as identifier for the matching `call` action. Finally, we add an identifier to the pair of `call` and `return` actions for external function calls. External function calls are not subject to the ambiguity, since they never occur nested in one another. We arbitrarily choose `0` as the identifier for external function calls.

We have solved the last remaining ambiguity and with that, our mCRL2 specification is finished. Refer to Appendix A for the complete implementation details of the specification.

Chapter 5

Translation of Dyno to μ-calculus

In previous chapters we gave an overview of Dyno and showed how a model can be extracted from an Imp program, representing its state space. This chapter focuses on translating Dyno patterns into μ-calculus formulas that operate on these extracted models. The translation is designed to match the intuitive meanings given to Dyno patterns in Chapter 3. In Section 5.1 we show how each of Dyno's pattern constructs can be translated to corresponding μ-calculus formulas through examples. A complete, formal, translation of Dyno to μ-formulas is given in Section 5.2. In Section 5.3 we wrap up by discussing our implementation of Dyno using the Spoofax language workbench.

5.1 Translating Dyno operators to μ-calculus formulas

In the following subsections we go through examples of each of Dyno's constructs and show how they can be translated to μ-calculus formulas. The aim is to provide μ-formulas that are true for the extracted model of a program if and only if the corresponding Dyno pattern matches the corresponding program.

5.1.1 Concrete syntax

The syntax for concrete syntax patterns is `{{ s }}`, where s is any Imp statement. The following is an example of such a pattern.

```
1 {{ 0 }}
```

This syntactic pattern matches the constant 0. We can represent it using the following μ-calculus formula.

$$\langle \texttt{start\_expr}(e) \rangle \exists v \in \texttt{Val}. \langle \texttt{end\_expr}(e, v) \rangle \top$$
$$\text{where } e := \texttt{constant(NumV(0))}$$

Since we are only matching syntax, we do not care about the value of the expression. We model this using an existential quantification to capture the value of the expression. Similar is the next pattern that matches a variable x.

```
1 {{ x }}
```

With the following μ-calculus formula.

$$\langle \texttt{start\_expr}(e) \rangle \exists v \in \texttt{Val}. \langle \texttt{end\_expr}(e, v) \rangle \top$$
$$\text{where } e := \texttt{variable(x)}$$

In general, translations of atomic expressions follow the exact same structure with different values for e. Next, we show an example of an expression with sub-expressions, the addition operator.

```
1 {{ x + y }}
```

In this case, we have to represent the expression itself, as well as its sub-expressions, in the μ-formula.

$$
\begin{aligned}
&\langle \texttt{start\_expr}(e)\rangle \\
&\quad \langle \texttt{start\_expr}(e_l)\rangle \exists v_l \in \texttt{Val}.\langle \texttt{end\_expr}(e_l,v_l)\rangle \\
&\quad \langle \texttt{start\_expr}(e_r)\rangle \exists v_r \in \texttt{Val}.\langle \texttt{end\_expr}(e_r,v_r)\rangle \\
&\quad \exists v \in \texttt{Val}.\langle \texttt{end\_expr}(e,v)\rangle \top \\
\text{where } e &:= \texttt{binop(Add,}e_l\texttt{,}e_r\texttt{)} \\
e_l &:= \texttt{variable(x)} \\
e_r &:= \texttt{variable(y)}
\end{aligned}
$$

Similarly, for statements we should translate the statement itself, as well as its sub-expressions and sub-statements. Consider the assignment statement for example.

```
1 {{ x = 0 }}
```

We translate it to μ-calculus as follows.

$$
\begin{aligned}
&\exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle\langle \texttt{start\_expr}(e)\rangle \\
&\quad \exists v \in \texttt{Val}.\langle \texttt{end\_expr}(e,v)\rangle \\
&\quad \langle \texttt{end\_stmt}(s,i)\rangle \top \\
\text{where } s &:= \texttt{assign(x,}e\texttt{)} \\
e &:= \texttt{constant(NumV(0))}
\end{aligned}
$$

We use an existential quantification to catch the numeric identifier of the assignment. It does not matter what it is, as long as it is the same for the `start_stmt` and `end_stmt` actions.

Next, we look at an example of a pattern that matches a function call syntactically.

```
1 {{ foo(x) }}
```

The pattern matches a call to function `foo` with the variable `x` as its only parameter. We first desugar the function call into an assign function call, such that it becomes `dummy = foo(x)`. The μ-calculus formula to match this (desugared) function call is:

$$
\begin{aligned}
&\exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle\langle \texttt{start\_expr}(e)\rangle\langle \texttt{end\_expr}(e)\rangle \\
&\quad \exists v_{arg} \in \texttt{Val}.\langle \texttt{call(foo,[}v_{arg}\texttt{],}i\texttt{)}\rangle \mu X.(\\
&\quad\quad \exists v_r \in \texttt{Val}.\langle \texttt{ret(foo,}v_r\texttt{,}i\texttt{)}\rangle\langle \texttt{end\_stmt}(s,i)\rangle \top \\
&\quad) \vee \forall v_r \in \texttt{Val}.[\overline{\texttt{ret(foo,}v_r\texttt{,}i\texttt{)}}]X \wedge \langle \overline{\texttt{ret(foo,}v_r\texttt{,}i\texttt{)}}\rangle \top \\
\text{where } s &:= \texttt{call(dummy,foo,}e\texttt{)} \\
e &:= \texttt{variable(x)}
\end{aligned}
$$

We not only include `start_stmt` and `end_stmt`, but also `call` and `ret` actions. This is not strictly necessary, but it is more consistent with the formula for function call patterns in Section 5.1.4. We use a fixed point, because we neither know nor care about what the internal computation of the function looks like. Since we are not capturing the return value, we do not care what exact value it takes on and therefore abstract it away using an existential quantification. On the right-hand side of the disjunction we use universal quantification due to the action in the modality being negated. A box modality ($[\overline{\texttt{ret(foo,}v_r\texttt{,}i\texttt{)}}]X$) is used here, because multiple internal computation paths might exist, for example if a call to an external function occurs. The box modality ensures that eventually the end of the function call is reached on *all* branches. Additionally, the subformula $\langle \overline{\texttt{ret(foo,}v_r\texttt{,}i\texttt{)}}\rangle \top$ is needed to ensure

the formula does not trivially hold due to the box modality when a `ret(foo,`v_r`,i)` action is encountered.

The translation of if-statements is more complex, because we do not always know a priori whether the then-branch or the else-branch will be taken during execution. An example of a concrete syntax pattern matching an if-statement is the following.

```
{{ if (x) x else y }}
```

The trick we use to devise a pattern that matches the if-statement, regardless of whether the then-branch or the else-branch is taken during computation, is as follows. First, we capture the value of the value of the condition. Then, we use a disjunction to distinguish between the 'true' and 'false' case and continue the pattern accordingly. For the if-statement above this leads to the following μ-formula.

$$
\begin{aligned}
&\exists e \in \texttt{Expr}, i \in \texttt{Int}.\langle \texttt{start\_stmt(ite(}e\texttt{,}s_1\texttt{,}s_2\texttt{),}i\texttt{)}\rangle\langle \texttt{start\_expr(}e\texttt{)}\rangle \mu X.(\\
&\quad \exists v \in \texttt{Val}.\langle \texttt{end\_expr(}e\texttt{,}v\texttt{)}\rangle \text{val}(e\texttt{==}x) \wedge \\
&\quad (\text{val}(v\texttt{==true}) \wedge \phi_1\langle \texttt{end\_stmt(ite(}e\texttt{,}s_1\texttt{,}s_2\texttt{),}i\texttt{)}\rangle\top \vee \\
&\quad \text{val}(v\texttt{==false}) \wedge \phi_2\langle \texttt{end\_stmt(ite(}e\texttt{,}s_1\texttt{,}s_2\texttt{),}i\texttt{)}\rangle\top) \\
&) \vee \forall v \in [\overline{\texttt{end\_expr(}e\texttt{,}v\texttt{)}}]X \wedge \langle \overline{\texttt{end\_expr(}e\texttt{,}v\texttt{)}}\rangle\top \\
\text{where } &x := \texttt{variable(x)}, s_1 := \texttt{expr(}x\texttt{)}, s_2 := \texttt{expr(variable(y))} \\
&\phi_1 := \exists i_1 \in \texttt{Int}.\langle \texttt{start\_stmt(}s_1\texttt{,}i_1\texttt{)}\rangle \ldots \langle \texttt{end\_stmt(}s_1\texttt{,}i_1\texttt{)}\rangle \\
&\phi_2 := \exists i_2 \in \texttt{Int}.\langle \texttt{start\_stmt(}s_2\texttt{,}i_2\texttt{)}\rangle \ldots \langle \texttt{end\_stmt(}s_2\texttt{,}i_2\texttt{)}\rangle
\end{aligned}
$$

A lot is going on in this formula. Subformulas ϕ_1 and ϕ_2 simply match the then-branch and false-branch of the if-statement, respectively. Using val(v`==true`) and val(v`==false`) we check whether the condition value, which is captured with v, is `true` or `false`. Depending on which case holds either the then-branch or the else-branch subsequently needs to match. The last important detail is that we replaced the if-statement condition with an existentially quantified variable e. We then assert that e needs to be the same as the actual if-statement condition x. While in this case we could have gotten away with directly using the real if-statement condition instead of substituting it, this formulation is more general. In particular when the condition is a value capture expression this substitution is necessary to preserve the value capture.

The last concrete syntax pattern we discuss is one that matches a while loop. Take the following example.

```
{{ while(true) {} }}
```

This pattern matches an infinite while loop with an empty body. To implement this pattern as a μ-calculus formula, the formula should somehow match the loop structure. The start of the pattern should match the while loop syntactically using a `start_stmt` action. Since the loop condition is evaluated at least once for every while loop, we also match the condition expression once. After that, we use a fixed point to match consequent repetitions of the loop body followed by the loop condition, until possibly an `end_stmt` action that closes the loop is

found. This results in the following pattern.

$$
\begin{aligned}
&\exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle \langle cond\rangle\, \nu X.(\\
&\quad \langle \texttt{end\_stmt}(s,i)\rangle \top\\
&\quad \vee \langle body\rangle\langle \tau\rangle\langle cond\rangle X\\
&)\\
\text{where } & e_c := \texttt{constant(BoolV(true))}\\
& s_b := \texttt{blk([])}\\
& s := \texttt{while}(e_c)\ s_b\\
& \langle cond\rangle := \langle \texttt{start\_expr}(e_c)\rangle\langle \texttt{end\_expr}(e_c)\rangle\\
& \langle body\rangle := \exists i_b \in \texttt{Int}.\langle \texttt{start\_stmt}(s_b,i_b)\rangle\langle \texttt{end\_stmt}(s_b,i_b)\rangle
\end{aligned}
$$

Note the use of the greatest fixed point (ν) here, instead of the least fixed point (μ) that we use for most patterns. The reason for choosing the greatest fixed point is precisely such that infinite loops can be matched. If the least fixed point were used instead, the formula would fail to match an infinite loop of the form shown in the pattern. This is because the least fixed point is allowed to 'pass through' the fixed point variable (X) only a finite amount of times, before an `end_stmt` action is reached. In the case of an infinite loop this never happens, so using the least fixed point would not result in a match. On the other hand, the above formula with the greatest fixed point does correctly match infinite loops, because it *is* allowed to pass through X an infinite amount of times.

Between the body and the condition there is a τ-action. This is necessary because it is present in the same location in the representations of while loops in LTSs.

5.1.2 Metavariables

Metavariables are used to capture arbitrary syntax or runtime values. Consider the following pattern with an expression metavariable.

```
1 var @e: expression
2 {{ x = @e }}
```

This pattern matches any assignment to `x`, regardless of the right-hand side expression.

At the end of Section 4.3.3 we already hinted how metavariables can be translated to μ-calculus. Here we give the μ-formula of the above pattern as an example of how to translate a metavariable and its declaration.

$$
\begin{aligned}
&\exists e \in \texttt{Expr}, i \in \texttt{Int}.\langle \texttt{start\_stmt(assign(x,}e\texttt{),}i\texttt{)}\rangle\langle \texttt{start\_expr}(e)\rangle \mu X.(\\
&\quad \exists v \in \texttt{Val}.\langle \texttt{end\_expr}(e,v)\rangle\langle \texttt{end\_stmt(assign(x,}e\texttt{),}i\texttt{)}\rangle \top\\
&\quad \vee \forall v \in \texttt{Val}.[\overline{\texttt{end\_expr}(e,v)}]X \wedge \langle \overline{\texttt{end\_expr}(e,v)}\rangle \top\\
&)
\end{aligned}
$$

The declaration is represented as an existential quantification. The metavariable is represented as a fixed point formula of a form that is probably familiar by now.

Lastly, we note that identifier and value metavariables can occur anywhere an expression can occur, and expression metavariables can occur anywhere a statement can occur. We handle this by 'upcasting' metavariables to the expected type. Take the following identifier metavariable as an example.

```
1 var @x: identifier
2 y = @x
```

This is a valid pattern, even though one might expect a metavariable of type `expression` in this position. To ensure the μ-calculus translation is correct, we wrap the metavariable in a `variable` node during compilation.

5.1.3 Ellipsis

The ellipsis pattern can be used to iterate over arbitrary syntax and computation until a pattern of interest is found. We revisit an example from Section 3.2.1 of the diamond ellipsis.

```
<...>
{{ foo() }}
```

And a similar example of the box ellipsis.

```
[...]
{{ foo() }}
```

The μ-calculus translations for the ellipsis variants are quite similar. First we give the translation of the diamond (exists) variant. Where, for brevity we elide some of the actions that correspond to the function call statement.

$$\begin{aligned} & \mu X.\phi \vee \langle\top\rangle X \\ \text{where } s &:= \texttt{call(dummy,foo,[])} \\ & \phi := \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle \ldots \langle \texttt{end\_stmt}(s,i)\rangle \end{aligned}$$

The translation of the box (forall) variant is as follows.

$$\begin{aligned} & \mu X.\phi \vee [\top]X \wedge \langle\top\rangle\top \\ \text{where } s &:= \texttt{call(dummy,foo,[])} \\ & \phi := \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle \ldots \langle \texttt{end\_stmt}(s,i)\rangle \end{aligned}$$

Compared to the diamond variant, we exchange the diamond modality for a box modality and add a diamond modality to exclude a trivial match in case of deadlock.

5.1.4 Function call patterns

Function call patterns are used to match occurrences of function calls. The following is a simple example of a pattern that matches a call to a function named `foo`.

```
foo()
```

The μ-calculus translation of the call pattern is as follows.

$$\begin{aligned} & \exists i \in \texttt{Int}.\langle \texttt{call(foo,[],}i\texttt{)}\rangle \mu X. \\ & \quad \exists v \in \texttt{Val}.\langle \texttt{ret(foo,}v\texttt{,}i\texttt{)}\rangle\top \vee \forall v \in \texttt{Val}.[\overline{\texttt{ret(foo,}v\texttt{,}i\texttt{)}}] \wedge \langle\overline{\texttt{ret(foo,}v\texttt{,}i\texttt{)}}\rangle\top \end{aligned}$$

Note that the μ-calculus translation of the concrete syntax version of the function call pattern is essentially a function call pattern wrapped with `start_stmt` and `end_stmt` actions.

5.1.5 Value capture

Dyno offers a collection of value capture constructs to be able to capture the value of any expression and the return value of any function call. In particular, we have:

- The expression value capture pattern: e `->` v, that can be used in place of any expression to capture its value.
- A capture version of the function call pattern: id`(`[v`,`]`*)` `->` v, used to capture the return value of a function call.
- Capture versions of the concrete syntax function call patterns, used to capture return values.

- Call statement capture: `{{ id([e,]*) -> v }}`
- Call assign statement capture: `{{ id = id([e,]*) -> v }}`

An example of the use of an expression value capture pattern is the following.

```
1 var @e: expression
2 <...>
3 {{ x->0 }}
```

This pattern asserts that there is an `x` expression that evaluates to `0` at some point in the program. The next pattern asserts that there is a function that returns the same value as its input parameter.

```
1 var @f: identifier
2 var @v: value
3 <...>
4 @f(@v) -> @v
```

To compile the value capture patterns, we take the compilation of the non-capture variant of the pattern and replace the existentially quantified value variable by the actual value as specified in the capture pattern. Consider the translation of the expression capture pattern above as an example.

$$\mu X.\langle \texttt{start\_expr(variable(x))} \rangle \langle \texttt{end\_expr(variable(x),NumV(0))} \rangle \top \vee \langle \top \rangle X$$

5.1.6 Assertion

This assertion pattern can be used to assert restrictions on values captured with metavariables using Imp expression syntax. Consider the following example.

```
1 var @e: expression
2 var @v: value
3 x = @e->@v
4 assert @v > 0
```

The asserted expression must evaluate to a boolean value, otherwise it is invalid. Translating assertions to μ-calculus formulas is relatively straightforward. Consider the following translation of the above pattern.

$$\begin{aligned} &\exists e \in \texttt{Expr}, v \in \texttt{Val}. \exists i \in \texttt{Int}. \langle \texttt{start\_stmt}(s,i) \rangle \langle \texttt{start\_expr}(e) \rangle \\ &\quad \langle \texttt{end\_expr}(e,v) \rangle \langle \texttt{end\_stmt}(s,i) \rangle \\ &\qquad \text{val}(\texttt{get\_bool(eval}(prop,\texttt{empty\_env},\texttt{empty\_env}))) \\ &\text{where } prop := \texttt{binop(Gt},v,\texttt{constant(NumV(0)))} \end{aligned}$$

The last part, val(...), corresponds to the assertion from the pattern. It evaluates the supplied binary operation, and if $prop$ evaluates to `BoolV(true)`, val(`get_bool(BoolV(true))`) becomes $\top$. Otherwise, if $prop$ evaluates to `BoolV(false)`, it becomes $\bot$ causing the entire formula to evaluate to $\bot$.

5.1.7 Negation

The final pattern we discuss is the negation pattern. Negation in Dyno functions exactly like negation in propositional logic or μ-calculus. If a Dyno pattern matches a program, then its negation does not match, and vice versa. For example, the following pattern matches a program that does not contain a call to `print(3)`.

```
1 !<...>
2 print(3)
```

Negation is simply implemented using the negation operator from μ-calculus. Thus, the translation of the above formula would be the translation of `<...> print(3)` prepended by a negation.

5.2 A complete translation of Dyno to μ-calculus

In the previous section we have seen examples of translations from Dyno patterns into μ-calculus formulas. In this section we formalize this by giving a complete overview of the systematic translation into μ-calculus. We give the translation using a polymorphic translation function '$[\![\cdot]\!]_\phi$', formally defined as follows.

Definition 5.1. Given a μ-formula ϕ, the polymorphic translation operator $[\![\cdot]\!]_\phi$ maps an arbitrary item to a μ-calculus formula containing ϕ as sub-formula. Additionally, $[\![\cdot]\!]$ is used as an alias for $[\![\cdot]\!]_\top$, i.e. the translation where the sub-formula is $\top$.

Throughout this section, we give equations that define the translation function on Dyno patterns. We start by defining the following utility operation for translating lists of items.

Definition 5.2. An operation that defines how to translate a list of generic items into a formula. Here, `[]` denotes the empty list and `[`s`|`ss`]` is a list with s as its first element and ss as its tail.

$$[\![\texttt{[]}]\!]_\phi = \phi$$
$$[\![\texttt{[}s\texttt{|}ss\texttt{]}]\!]_\phi = [\![s]\!]_{[\![ss]\!]_\phi}$$

In Figure 5.1 the translation of top-level Dyno patterns is defined. First, it translates the metavariable declarations using nested existential quantifiers. The pattern p is then translated as the sub-formula of the inner existential quantifier.

$$[\![mv^*\ \ p]\!] = [\![mv^*]\!]_{[\![p]\!]_\top}$$
$$[\![\texttt{var @}x\ \texttt{:}\ \tau]\!]_\phi = \exists x \in [\![\tau]\!]^{\text{Imp}}.\phi$$

Figure 5.1. Top-level translation of a Dyno pattern.

The operator $[\![\cdot]\!]^{\text{Imp}}$ translates Imp statements, expressions, values and types into mCRL2 data types. We do not formally define the operator in this section, but an example of its application is shown in the following example.

Example 5.1. Example of applying $[\![\cdot]\!]^{\text{Imp}}$ to the expression `x + 2`.

$$[\![\texttt{x + 2}]\!]^{\text{Imp}} = \texttt{binop(Add, variable(x), constant(NumV(2)))}$$

Translation of Dyno pattern constructs is shown in Figure 5.2. These translations are generalizations of those given in the examples of Section 5.1. Note that for the concrete syntax pattern we provide two equations. This is an attempt to communicate that if an expression statement is encountered at the top level of a concrete syntax pattern, the inner expression should be used instead of the expression statement.

In the definitions in this section some syntax is annotated with `expr`, `stmt`, `id` or `value` when it is ambiguous which is meant. For example, we use $e^{\texttt{stmt}}$ to distinguish an expression statement from an expression $e^{\texttt{expr}}$.

$$
\begin{aligned}
[\![\texttt{\{\{ } e^{\texttt{stmt}} \texttt{ \}\}}]\!]_\phi &= [\![e^{\texttt{expr}}]\!]_\phi \\
[\![\texttt{\{\{ } s \texttt{ \}\}}]\!]_\phi &= [\![s]\!]_\phi \\
[\![\texttt{<...>}]\!]_\phi &= \mu X.\phi \vee \langle\top\rangle X \\
[\![\texttt{[...]}]\!]_\phi &= \mu X.\phi \vee [\top] X \wedge \langle\top\rangle\top \\
[\![\texttt{assert } e]\!]_\phi &= \text{val}(\texttt{get\_bool(eval(}e'\texttt{,empty\_env,empty\_env))}) \wedge \phi \\
&\quad \text{where } e' := [\![e]\!]^{\textsc{Imp}} \\
[\![id\texttt{([}v\texttt{,]*)}]\!]_\phi &= \exists i \in \texttt{Int}.\langle\texttt{call(}id\texttt{,}vs\texttt{,}i\texttt{)}\rangle \\
&\qquad \mu X.(\exists v_r \in \texttt{Val}.\langle\texttt{ret(}id\texttt{,}v_r\texttt{,}i\texttt{)}\rangle\phi \\
&\qquad\quad \vee \forall v_r \in \texttt{Val}.[\overline{\texttt{ret(}id\texttt{,}v_r\texttt{,}i\texttt{)}}]X \wedge \langle\overline{\texttt{ret(}id\texttt{,}v_r\texttt{,}i\texttt{)}}\rangle\top) \\
&\quad \text{where } vs := [\![[v\texttt{,}]^*]\!]^{\textsc{Imp}} \\
[\![id\texttt{([}v\texttt{,]*) -> } v_r]\!]_\phi &= \exists i \in \texttt{Int}.\langle\texttt{call(}id\texttt{,}vs\texttt{,}i\texttt{)}\rangle \\
&\qquad \mu X.(\langle\texttt{ret(}id\texttt{,}v'_r\texttt{,}i\texttt{)}\rangle\phi \vee [\overline{\texttt{ret(}id\texttt{,}v'_r\texttt{,}i\texttt{)}}]X \wedge \langle\overline{\texttt{ret(}id\texttt{,}v'_r\texttt{,}i\texttt{)}}\rangle\top) \\
&\quad \text{where } \begin{cases} vs := [\![[v\texttt{,}]^*]\!]^{\textsc{Imp}} \\ v'_r := [\![v_r]\!]^{\textsc{Imp}} \end{cases} \\
[\![\texttt{!}p]\!]_\phi &= \neg[\![p]\!]_\phi \\
[\![p_1\ p_2]\!]_\phi &= [\![p_1]\!]_{[\![p_2]\!]_\phi}
\end{aligned}
$$

Figure 5.2. Translation of Dyno pattern constructs.

We further define the translation function by adding cases for concrete syntax statements in Figures 5.3 and 5.4. Before translation, we assume that an oracle has annotated all metavariables with a type, based on the metavariable declarations at the top of the pattern. The notation $\texttt{@}x[\tau]$ is used for metavariable $\texttt{@}x$, annotated with type τ. In practice, we implement this oracle as a type checking phase in the compilation pipeline.

Completing the translation of Dyno patterns, we define cases of the $[\![\cdot]\!]_\phi$ function for metavariables and expressions in Figures 5.5 and 5.6.

$$
\begin{aligned}
[\![e^{\texttt{stmt}}]\!]_\phi &= \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle [\![e^{\texttt{expr}}]\!]_{\langle \texttt{end\_stmt}(s,i)\rangle\phi} \\
&\quad \text{where } s := [\![e^{\texttt{stmt}}]\!]^{\textsc{Imp}} \\
[\![x \texttt{ = } e]\!]_\phi &= \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle [\![e]\!]_{\langle \texttt{end\_stmt}(s,i)\rangle\phi} \\
&\quad \text{where } s := [\![x \texttt{ = } e]\!]^{\textsc{Imp}} \\
[\![\texttt{global } x \texttt{ = } e]\!]_\phi &= \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle [\![e]\!]_{\langle \texttt{end\_stmt}(s,i)\rangle\phi} \\
&\quad \text{where } s := [\![\texttt{global } x \texttt{ = } e]\!]^{\textsc{Imp}} \\
[\![\texttt{while } (e) \ s]\!]_\phi &= \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle \phi_{cond} \\
&\quad \text{where } \begin{cases} \phi_{cond} := [\![e]\!]_{\phi_\nu} \\ \phi_\nu := \nu X.\langle \texttt{end\_stmt}(s,i)\rangle\phi \vee \phi_{loop} \\ \phi_{loop} := [\![s]\!]_{\langle\tau\rangle [\![e]\!]_X} \end{cases} \\
[\![x \texttt{ = } f\texttt{(}[e,]^*\texttt{)}]\!]_\phi &= \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle [\![e^*]\!]_{\phi_{call}} \\
&\quad \text{where } \begin{cases} s := [\![x \texttt{ = } f\texttt{(}[e,]^*\texttt{)}]\!]^{\textsc{Imp}} \\ v^* := \text{generate a unique identifier for each } e \in e^* \\ \phi_{call} := [\exists v \in \texttt{Val}.]^* [\![f\texttt{(}[@v[\texttt{value}],]^*\texttt{)}]\!]_{\langle \texttt{end\_stmt}(s,i)\rangle\phi} \end{cases} \\
[\![x \texttt{ = } f\texttt{(}[e,]^*\texttt{) -> } v_r]\!]_\phi &= \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle [\![e^*]\!]_{\phi_{call}} \\
&\quad \text{where } \begin{cases} s := [\![x \texttt{ = } f\texttt{(}[e,]^*\texttt{)}]\!]^{\textsc{Imp}} \\ v^* := \text{generate a unique identifier for each } e \in e^* \\ \phi_{call} := [\exists v \in \texttt{Val}.]^* [\![f\texttt{(}[@v[\texttt{value}],]^*\texttt{) -> } v_r]\!]_{\langle \texttt{end\_stmt}(s,i)\rangle\phi} \end{cases} \\
[\![\texttt{\{ } [s;]^* \texttt{ \}}]\!]_\phi &= \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s',i)\rangle [\![s^*]\!]_{\langle \texttt{end\_stmt}(s',i)\rangle\phi} \\
&\quad \text{where } s' := [\![\texttt{\{ } [s;]^* \texttt{ \}}]\!]^{\textsc{Imp}} \\
[\![\texttt{return } e]\!]_\phi &= \exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s,i)\rangle [\![e]\!]_{\langle \texttt{end\_stmt}(s,i)\rangle\phi} \\
&\quad \text{where } s := [\![\texttt{return } e]\!]^{\textsc{Imp}}
\end{aligned}
$$

Figure 5.3. Translation of concrete syntax statement patterns.

$$
\begin{aligned}
[\![\texttt{if } (e) \ s_\top \texttt{ else } s_\bot]\!]_\phi &= \exists e' \in \texttt{Expr}.\exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(s',i)\rangle \phi' \\
&\quad \text{where } \begin{cases} s' := [\![\texttt{if (@}e'[\texttt{Expr}]\texttt{) } s_\top \texttt{ else } s_\bot]\!]^{\textsc{Imp}} \\ \phi' := \langle \texttt{start\_expr}(e')\rangle \mu X. \\ \qquad \phi'' \vee \forall v \in \texttt{Val}.\overline{[\texttt{end\_expr}(e,v)]}X \wedge \overline{\langle \texttt{end\_expr}(e,v)\rangle}\top \\ \phi'' := \exists v \in \texttt{Val}.\langle \texttt{end\_expr}(e,v)\rangle \phi''' \\ \phi''' := (\text{val}(e' \texttt{ == } e) \wedge (\phi_\top \vee \phi_\bot)) \\ \phi_\top := \text{val}(v \texttt{ == BoolV(true)}) \wedge [\![s_\top]\!]_{\langle \texttt{end\_stmt}(s',i)\rangle\phi} \\ \phi_\bot := \text{val}(v \texttt{ == BoolV(false)}) \wedge [\![s_\bot]\!]_{\langle \texttt{end\_stmt}(s',i)\rangle\phi} \end{cases}
\end{aligned}
$$

Figure 5.4. Translation of the if statement concrete syntax pattern.

$$
[\![@x[\tau]]\!]_\phi = \begin{cases}
\exists i \in \texttt{Int}.\langle \texttt{start\_stmt}(x', i)\rangle \mu X. & \\
\quad (\langle \texttt{end\_stmt}(x', i)\rangle \phi & \text{if } \tau = \texttt{statement} \\
\quad\quad \vee \overline{[\texttt{end\_stmt}(x', i)]} X \wedge \overline{\langle \texttt{end\_stmt}(x', i)\rangle} \top) & \\
\\
\langle \texttt{start\_expr}(x')\rangle \mu X. & \\
\quad (\exists v \in \texttt{Val}.\langle \texttt{end\_expr}(x', v)\rangle \phi & \text{otherwise} \\
\quad\quad \vee \forall v \in \texttt{Val}.\overline{[\texttt{end\_expr}(x', v)]} X \wedge \overline{\langle \texttt{end\_expr}(x', v)\rangle} \top) &
\end{cases}
$$

$$
\text{where } x' := [\![@x[\tau]]\!]^{\text{Imp}}
$$

$$
[\![@x[\tau]\texttt{->}v]\!]_\phi = \langle \texttt{start\_expr}(x')\rangle \mu X.
$$

$$
(\langle \texttt{end\_expr}(x', v')\rangle \phi \vee \overline{[\texttt{end\_expr}(x', v')]} X \wedge \overline{\langle \texttt{end\_expr}(x', v')\rangle} \top)
$$

$$
\text{where } \begin{cases} x' := [\![@x[\tau]]\!]^{\text{Imp}} \\ v' := [\![v]\!]^{\text{Imp}} \end{cases}
$$

Figure 5.5. Translation of concrete syntax metavariables and metavariable capture.

$$
[\![x^{\texttt{expr}}]\!]_\phi = \langle \texttt{start\_expr}(e)\rangle \exists v \in \texttt{Val}.\langle \texttt{end\_expr}(e, v)\rangle \phi
$$

$$
\text{where } e := [\![x^{\texttt{expr}}]\!]^{\text{Imp}}
$$

$$
[\![x^{\texttt{expr}}\texttt{->}v_r]\!]_\phi = \langle \texttt{start\_expr}(e)\rangle\langle \texttt{end\_expr}(e, v_r)\rangle \phi
$$

$$
\text{where } \begin{cases} e := [\![x^{\texttt{expr}}]\!]^{\text{Imp}} \\ v'_r := [\![v_r]\!]^{\text{Imp}} \end{cases}
$$

$$
[\![v^{\texttt{expr}}]\!]_\phi = \langle \texttt{start\_expr}(e)\rangle \exists v_r \in \texttt{Val}.\langle \texttt{end\_expr}(e, v_r)\rangle \phi
$$

$$
\text{where } e := [\![v^{\texttt{expr}}]\!]^{\text{Imp}}
$$

$$
[\![v^{\texttt{expr}}\texttt{->}v_r]\!]_\phi = \langle \texttt{start\_expr}(e)\rangle\langle \texttt{end\_expr}(e, v'_r)\rangle \phi
$$

$$
\text{where } \begin{cases} e := [\![v^{\texttt{expr}}]\!]^{\text{Imp}} \\ v'_r := [\![v_r]\!]^{\text{Imp}} \end{cases}
$$

$$
[\![\odot e]\!]_\phi = \langle \texttt{start\_expr}(e')\rangle [\![e]\!]_{\exists v \in \texttt{Val}.\langle \texttt{end\_expr}(e', v)\rangle \phi}
$$

$$
\text{where } e' := [\![\odot e]\!]^{\text{Imp}}
$$

$$
[\![\odot e\texttt{->}v_r]\!]_\phi = \langle \texttt{start\_expr}(e')\rangle [\![e]\!]_{\langle \texttt{end\_expr}(e', v'_r)\rangle \phi}
$$

$$
\text{where } \begin{cases} e' := [\![\odot e]\!]^{\text{Imp}} \\ v'_r := [\![v_r]\!]^{\text{Imp}} \end{cases}
$$

$$
[\![e_l \oplus e_r]\!]_\phi = \langle \texttt{start\_expr}(e)\rangle [\![\texttt{[}e_l, e_r\texttt{]}]\!]_{\exists v \in \texttt{Val}.\langle \texttt{end\_expr}(e, v)\rangle \phi}
$$

$$
\text{where } e := [\![e_l \oplus e_r]\!]^{\text{Imp}}
$$

$$
[\![e_l \oplus e_r\texttt{->}v_r]\!]_\phi = \langle \texttt{start\_expr}(e)\rangle [\![\texttt{[}e_l, e_r\texttt{]}]\!]_{\langle \texttt{end\_expr}(e, v'_r)\rangle \phi}
$$

$$
\text{where } \begin{cases} e' := [\![e_l \oplus e_r]\!]^{\text{Imp}} \\ v'_r := [\![v_r]\!]^{\text{Imp}} \end{cases}
$$

Figure 5.6. Translation of concrete syntax expression patterns. We give a single equation for all unary operators $\odot \in \{\texttt{!}, \texttt{-}\}$, and another for binary operators $\oplus \in \{\texttt{\&\&}, \texttt{||}, \texttt{+}, \texttt{-}, \texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}\}$.

5.3 Implementation in Spoofax

This section discusses the implementation of the compiler used for automated translations of Dyno patterns and Imp programs. We do not provide full details, as the implementation is largely a direct implementation of the translations we defined previously. The compiler has been implemented in Spoofax (Kats and Visser 2010) and consists of the following parts.

- Implementation of Dyno, Imp and mCRL2 syntax in SDF3, which is Spoofax's domain-specific language (DSL) for specifying grammars.
- A strategy for compiling Dyno patterns to μ-calculus formulas, written in Stratego (Smits and Visser 2020), which is Spoofax's DSL for program transformation.
- A strategy for compiling Imp programs to mCRL2 specifications, based on the boilerplate we built in Chapter 4. The full boilerplate can be seen in Appendix A, where only lines 356-376 are program-specific.
- A type checker, implemented in Statix, that ensures metavariables are annotated with the correct types, based on their declarations.

To use Dyno, we provide syntax for combining an Imp program with a Dyno pattern in a single file with the `.dyn` extension. We call such a file a Dyno *specification*. The following is an example of a complete Dyno specification that checks a program for use-after-free bugs.

Example 5.2. Example of a Dyno specification that checks a program for occurrences of use-after-free bugs.

```
1  program
2    global
3      destroyed = false
4
5    public
6      fn access() -> void
7        if (!global destroyed) use()
8
9      fn destroy() -> void {
10       global destroyed = true;
11       free()
12     }
13
14   external
15     fn use() -> void
16     fn free() -> void
17
18 pattern
19   !<...>
20   free()
21   <...>
22   use()
```

A Dyno specification can be executed using the 'Run' option from a context menu within the Spoofax IDE. The result of pattern matching is written to an accompanying file. The context menu also contains options for generating an mCRL2 specification of the specified program or translating the specified pattern to a μ-calculus formula.

5.3.1 Correctness of the implementation

To ensure the correctness of our implementation, we have developed a suite of automated tests. These tests are designed to validate the behavior of the compiler and ensure that it adheres to the expected semantics of Dyno patterns.

Our testing strategy is twofold. First, we verify the correctness of individual operators and constructs. Each operator in Dyno is tested at least once to confirm its behavior matches the informal semantics described in earlier chapters. For concrete syntax patterns, we ensure that every type of Imp expression and statement is tested with both positive and negative cases. This approach aims to ensure that the compiler handles all Imp constructs correctly.

Second, we validate the correctness of more complex patterns and edge cases. For example, we test scenarios that distinguish between box and diamond ellipses, as well as scenarios requiring function call patterns instead of their concrete syntax equivalents. Additionally, we implement the examples from Chapter 3 as test cases to verify that the compiler handles practical patterns effectively.

To illustrate, consider the following two tests for verifying assignment matching. The first test checks a positive case where an assignment is correctly matched, while the second test ensures that an unmatched assignment is correctly identified as a negative case.

Example 5.3. A positive test for assignment matching.

```
test assign (positive) [[
  program
    public
      fn main() -> void {
        x = true
      }

  pattern
  <...>
  {{ x = true }}
]] run run-spec to "true"
```

Example 5.4. A negative test for assignment matching.

```
test assign (negative) [[
  program
    public
      fn main(x: bool) -> void {
        x
      }

  pattern
  <...>
  {{ x = true }}
]] run run-spec to "false"
```

Chapter 6

Limitations and future work

In this chapter we discuss some limitations of our current approach and how they might be improved upon in the future. Section 6.1 goes into the problem of infinite state spaces. In Section 6.2 we discuss the problem posed by the current lack of evidence in match results. Patterns that are currently not expressible in Dyno are discussed in Section 6.3, along with suggestions for operators that could improve expressiveness. Section 6.4 remarks that a logical next step would be to implement Dyno for a real programming language. Section 6.5 acknowledges the lack of a performance study in our work, which would be useful to do in future work. A different approach to semantic pattern matching, which trades precision for performance and possibly a termination guarantee is discussed in Section 6.6. In Section 6.7 we discuss program transformation as an area where semantic pattern matching might be applied. Finally, in Section 6.8 we remark that Dyno can also be interpreted as a DSL for property checking of programs, and how it could be augmented to be even more like a property language.

6.1 Infinite state spaces

We fundamentally cannot pattern match every program. In particular, programs with an infinite state space pose a problem. One source of infinite state spaces is when integer parameters are used as inputs to a program. We offer bound syntax to limit the input parameter space in such scenarios. However, even with a bounded input space, it is still possible to have programs with an infinite state space. An example of this is the following program.

```
1 x = 0;
2 while (true) {
3   x = x + 1;
4 }
```

Because the variable `x` is increased in every loop iteration of an infinite loop, the state space corresponding to this program never reaches a steady state. That is, a state after which every loop iteration is the same.

A similar problem occurs with infinite recursive functions. For example:

```
1 program
2   fn f() -> void
3     f()
```

The reason this program does not reach a steady state is because a new call frame is made for every function call. In this scenario a tail call optimization could offer a solution, as this would cause the state space to loop on the function call `f`, i.e. reach a steady state.

A more general solution, to both infinite loops and infinite recursive functions could be to let the user limit the amount of iterations of a loop or the size of the call stack. This would be a way to enforce termination, at the cost of changing the semantics of a program.

6.2 Lack of evidence in match results

The result of pattern matching currently is simply a boolean indicating whether a pattern matches the input program. If Dyno is to become a proper pattern matching tool it would need to report additional witness information of the match. For example, consider a pattern with a metavariable.

```
1 var @s: statement
2 <...>
3 {{ @s }}
```

We might be interested in finding out which statement(s) of the program cause(s) this pattern to be a match.

In the particular use case of program search, the user would want to know which part of a program matches the pattern. If Dyno can report positional AST information, it could be integrated into IDEs to highlight the syntax that matched a certain pattern.

6.3 Inexpressible patterns and additional operators

In previous chapters, we showed many examples of patterns that are expressible using Dyno. In this section we focus on patterns that are *not* expressible, but which one might expect to be expressible. That fact that these patterns are not expressible in the current version of Dyno indicates that there is room for improvement. Additionally, this section discusses examples of operators that could be added to the pattern language to make it more expressive.

6.3.1 Use-after-free with reallocation

In Section 3.3 we have seen an example of a use-after-free pattern that is expressible in Dyno. The pattern is defined as follows.

```
1 !<...>
2 free()
3 <...>
4 use()
```

Though this pattern is sufficient for the example program we used in Section 3.3, it might raise false positives when a reallocation happens after the free has occurred. Consider the following program.

```
1  global
2    destroyed = false
3
4  public
5    fn init() -> void
6      if (global destroyed) {
7        alloc();
8        global destroyed = false
9      }
10
11   fn access() -> void
```

```
12     if (!global destroyed) use()
13
14   fn destroy() -> void {
15     global destroyed = true;
16     free()
17   }
18
19 external
20   fn alloc() -> void
21   fn use() -> void
22   fn free() -> void
```

In the above program the resource can be reallocated, in which case the use-after-free pattern falsely reports that a use-after-free bug appears. To fix the pattern, such that it no longer reports a false positive, we need a way to restrict the ellipsis pattern. The following two sections discuss two patterns that achieve this.

6.3.2 The guarded ellipsis pattern

The problem of false positives for the use-after-free pattern in the previous section can be solved if we could limit the syntax that is allowed to occur between the `free` call and the `use` call. An example of a construct that could achieve this is the guarded ellipsis pattern. Using guarded ellipsis the improved pattern would look as follows.

```
1 !<...>
2 free()
3 <...> where !alloc()
4 use()
```

The μ-calculus translation of this particular guarded ellipsis might look as follows.

$$\mu X.\phi \vee (\langle\top\rangle X \wedge [\texttt{call(alloc,[}v\texttt{])}]\bot)$$

Where ϕ is a placeholder for the remainder of the pattern.

6.3.3 Reflexive transitive closure

An alternative to the guarded ellipsis pattern is to add a reflexive transitive closure pattern to the language. Similar to the star operator ('$\star$') from regular expressions, its syntax could be defined follows.

$$(p)\star$$

The meaning of this pattern is "pattern p repeated zero or more times." It could be translated to μ-calculus as follows.

$$\mu X.\phi \vee (\llbracket p \rrbracket_\top \wedge \langle\top\rangle X)$$

Where $\llbracket p \rrbracket_\top$ is the translation of the inner pattern and ϕ represents the remaining formula. We believe the reflexive transitive closure could greatly increase the expressiveness of Dyno, but we have not tested it properly, so we leave it to future work to further explore this pattern. Note that, depending on the desired semantics, the pattern could either be implemented using the least fixed point (as in our example translation) or the greatest fixed point. Or alternatively, multiple versions can be implemented leaving the choice of fixed point operator up to the user.

Using the reflexive transitive closure we can implement the use-after-free example from above as follows.

```
var @v: value
!<...>
free(@v)
(!alloc(@v))*
use(@v)
```

6.3.4 Scoped patterns

Currently, Dyno does not offer the ability to express scoped or nested patterns within if statements, while loops or block statements. To get an idea of what a *scoped pattern* might look like, consider the following example.

```
{{
  if (x) [
    <...> {{ print(x) }}
  ] else [
    !<...> {{ print(x) }}
  ]
}}
```

This imaginary example matches an if statement that has a `print(x)` statement in the true branch and does not have a `print(x)` statement in the false branch. The blocks of the if statement's branches have been replaced by a scope pattern, denoted using square brackets (`[` p `]`). The idea of a scope pattern is that an arbitrary Dyno pattern occurs within a given scope. In this case the scope would be the body of the if statement.

A potentially useful example of a scoped pattern would be one that finds a while loop for which a certain invariant holds. For example:

```
var @e: expression
{{
  while (@e) [
    var @v: value
    <...> {{ x->@v }}
    assert @v < 10
  ]
}}
```

This pattern finds a while loop where a variable `x` occurs, which is always smaller than `10`. Note that the pattern also requires metavariables to be declarable within a scope, otherwise the value would have to be exactly the same for each iteration.

Generally, scoped patterns would allow for a way to limit a certain sub-pattern to occur within a specific scope. The scope could be a function call, an if statement, while loop or block statement.

6.4 Application to a real programming language

In our primitive exploration of semantic pattern matching we have kept things simple by using a toy object language. It would be a logical next step to apply the technique to a real-world programming language. In doing so performance might be a growing concern, we discuss this in the next section. If indeed performance turns out to pose a problem, the approach we mention in Section 6.6 might be fruitful. The section suggests trading precision for smaller state spaces using over-approximation.

6.5 Performance study

In this thesis we have not focused on the performance of pattern matching with Dyno. Since we focus on an experimental scenario with a toy language, performance has not been our greatest concern. However, when moving on to more practical applications of our work, performance does become of interest.

Future work could look into the asymptotic complexity of pattern matching based on semantic models combined with model checking. To get an idea of the overhead of the semantic approach to non-semantic approaches, it would also be interesting to see an experimental study of the performance of Dyno patterns on programs, with a comparison to non-semantic alternatives.

Finally, as a coarse lower bound on performance, we do note that the state spaces extracted from Dyno programs are generally proportional to the runtime of the underlying object program. This is because during state space generation, we evaluate the actual runtime semantics of a program.

6.6 A different approach to the parameter space problem

During the development of Dyno we have explored different approaches to tackling the problem of state space explosion due to unknown input parameters. An alternative approach is to use over-approximation. In this approach, input parameters are modeled using approximation of their value space. The coarsest approximation would be to use an `any` value to represent all possible values of a parameter. Stricter bounds on a parameter's value can be established if it occurs in the condition of a branching construct.

Modified semantics of the object language have to be used to deal with approximated values. These semantics should, for example, specify what happens when operators are applied to one or more `any` values. Furthermore, if a termination guarantee is desired for pattern matching, loops and recursive calls have to be approximated as well. The technique we just described is essentially abstract interpretation (P. Cousot and R. Cousot 1976), a technique that uses over-approximation to reason about dynamic properties of programs while guaranteeing termination in finite time, but compromising on completeness.

Instead of over-approximation, in this thesis we use under-approximated state spaces of programs as the basis for pattern matching. Under-approximation has a significant advantage: it is simpler to implement, since the actual, non-approximated, object language semantics can be used. On top of this, it seems a natural fit with mCRL2, as, for example, unknown input parameters can easily be modeled using `sum` operators over data types.

That being said, a notable advantage of over-approximation is that it could lead to much smaller state spaces, likely leading to better performance. We therefore encourage further experimentation with over-approximation based semantic pattern matching.

6.7 Program transformation

A motivating use-case for pattern matching is as a necessary precursor to program transformation. Program transformation tools use patches expressed in a pattern language to automatically transform programs on a large scale. Therefore, a natural question to ask is whether Dyno can be applied in the context of program transformation. From a syntactic perspective, this requires adding syntax for expressing patches. For example, we could implement a *patch pattern*, as seen work by Miljak, Bach Poulsen, and Corvino (2024). For Dyno the patch pattern could be of the following form:

$$s \texttt{ => } s$$

Where s on the left-hand side is the concrete syntax to match and s on the right-hand side is the syntax that should replace it.

However, to be able to implement transformations, a method is needed to extract AST location information and metavariable bindings from pattern match results. As we have discussed in Section 6.2, this is not currently supported. Therefore, improving Dyno such that it does indeed report match evidence is necessary if it is to be used in the context of program transformation.

6.8 Dyno as a property language

We have focused on Dyno as a pattern language, motivated by the pattern matching use case. An alternative perspective is to view Dyno as a DSL to specify properties about programs using intuitive syntax. In this sense, it is essentially a layer of sugar on top of μ-calculus, specifically for the context of verifying properties about programs using concrete syntax.

When Dyno is used as a property language it might be fruitful to add additional logical operators to the language, such as conjunction and disjunction. We have experimented with such operators as pattern constructs. Implementing them is surprisingly straightforward. When conjunction and disjunction are defined at the pattern level, they can simply be implemented by translating them to their μ-calculus equivalents and continuing with copies of the remaining pattern on both sides of the conjunction or disjunction.

However, as we have not tested these constructs thoroughly, we leave it as future work to formally extend Dyno with extra operators and explore Dyno as a property language.

Chapter 7

Related work

This chapter discusses research efforts that are in some way related to this thesis. The sections are roughly organized in decreasing order of relevance. In Section 7.1 we discuss an existing tool that offers a pattern matching language similar to Dyno. Although it is not based on dynamic semantics, as a state-of-the-art tool in pattern matching it has been of great influence to our work. Section 7.2 discusses the model extraction technique that we base our Dyno implementation on. In Section 7.3 we mention a paper about concrete syntax metapatterns, from which we, among other things, borrow the terminology "concrete syntax patterns". Lastly, we briefly touch on abstract interpretation in Section 7.4.

7.1 Coccinelle

Coccinelle (Lawall and Muller 2018) is a transformation system developed with the main purpose of facilitating large scale collateral evolutions in systems code. At the core of Coccinelle lies a pattern language called SmPL. SmPL can be used to specify patches that match desired parts of a program and apply transformations at indicated locations. Consider the following example of a Coccinelle patch.

Example 7.1. A Coccinelle patch written in SmPL. A line starting with a - indicates that the statement or expression after it should be removed. Likewise, a line starting with + indicates the addition of syntax. Metavariables are declared in a header, delimited by `@@ @@`.

```
1 @@
2 identifier x;
3 @@
4 int x = 0;
5 ...
6 - print(x);
7 + print(x+1);
```

There are two fundamental differences between SmPL and our language, Dyno.

- SmPL pattern matching is control flow based, while Dyno pattern matching is based on a model that takes dynamic semantics into account.
- SmPL is not only capable of pattern matching, but also allows expressing transformations of syntax.

A more practical difference is that Coccinelle is implemented to work on a real programming language: C. It has been in development for well over a decade and supports many features that are useful in practice, such as support for scripting.

What SmPL and Dyno have in common is that they are both compiled to a modal logic and use a model checker to perform pattern matching. To implement SmPL, Brunel et al. (2009) propose CTL-VW as an extension of the better known temporal logic CTL. In the same paper it is shown how SmPL can be translated into CTL-VW primitives. One of the extensions in CTL-VW is existential quantification, which is necessary for the implementation of metavariables. The other is the collection of witnesses during model checking. In case of a match, a witness contains evidence in the form of metavariable bindings. These are used to apply transformations correctly.

We borrow the idea of compiling a pattern language to a modal logic from Coccinelle. However, instead of designing a custom logic, or adopting CTL-VW, we chose to use the modal μ-calculus. With its fixed points and quantification operators it has all the expressivity needed for implementing Dyno. A benefit of choosing μ-calculus is that we can use the mCRL2 model checker, instead of having to design our own.

7.1.1 Detecting use-after-free using Coccinelle

As an example of how Dyno patterns are more precise than semantic patching language (SmPL), we once again turn to the use-after-free pattern. In SmPL we can express the use-after-free pattern as follows.

```
@@
identifier x;
@@
free(x)
...
access(x)
```

This pattern catches the following use-after-free bug.

```
alloc(x);
free(x);
access(x);
```

However, Coccinelle fails to catch other occurrences of the bug, such as:

```
alloc(x);
y = x;
free(x);
access(y);
```

This bug is missed by Coccinelle because `x` and `y` are not syntactically equivalent. The following example reveals another scenario where Coccinelle.

```
alloc(x);
freed = false;
free(x);
freed = true;
if (!freed) {
  access(x);
}
```

Since Coccinelle is not aware that the true branch of the if statement is unreachable, it would falsely report the occurrence of a use-after-free bug. While the SmPL pattern could be updated to work for the above examples, doing so for generic use-after-free occurrences quickly becomes infeasible. This is a fundamental problem of syntactic pattern matching. As we have shown in Section 3.3.2, a semantic pattern matching language can be used to find use-after-free occurrences more precisely in a concise manner.

7.2 Semi-automatic extraction of formal models

Spaendonck (2024) proposes SSTraGen, a semi-automatic tool for the extraction of formal models from C++ classes. In Section 4.1 we use their technique for model extraction as the first step of obtaining a model that can be used for pattern matching.

SSTraGen roughly works as follows. Input C++ classes are parsed to an AST, which is in turn translated to a simpler language SCPP. Each class is then either transformed into an mCRL2 process equation or abstracted away as a stub process. The stub process over-approximates a class interface, by allowing any value to be returned from its methods. The return values of stubbed methods can also be bounded by the user. The user indicates one class as the top-level interface, of which the methods are the entry points of the process. Like the return values of stubbed methods, the parameters of entry point methods can be bounded. Finally, parallel composition of the resulting processes yields a model representing the (under-approximated) state space of a program. This model can be used for verification using μ-calculus formulas.

Both in concept and implementation there are similarities with our work. Their work also concerns the dynamic semantics of an object program. Furthermore, they also translate programs into mCRL2 specifications in order to generate LTSs of programs and to be able to verify properties about them using μ-calculus.

Although there is significant overlap, the goal of their work is different from ours. We do not aim for extraction of a model from code for verifying arbitrary μ-calculus properties. Instead, we provide a pattern matching DSL where concrete syntax can be used to express dynamic patterns of programs. Facilitating concrete syntax in behavioral patterns comes with its own set of challenges, which is one of the main concerns of this thesis.

7.3 Concrete syntax metapatterns

Miljak, Bach Poulsen, and Corvino (2024) present a technique for implementing concrete syntax metapatterns. Similar to our work, they provide a pattern language based on concrete syntax, which means that object language syntax can be used as a primitive when writing patterns. This makes for much more readable and intuitive patterns, when compared to patterns based on abstract syntax. By employing a black-box parser technique, the need for implementing a custom parser is circumvented. This is particularly advantageous for languages with complex syntax like C++. Like Coccinelle, they support program transformation through a patch pattern.

7.4 Abstract interpretation

Abstract interpretation, pioneered in the work of P. Cousot and R. Cousot (1976), is a technique used to analyze dynamic properties of programs statically. In Section 6.6 we mention abstract interpretation as an alternative basis for semantic pattern matching. The benefit of abstract interpretation is that it uses approximated semantics, which results in a smaller state space during analysis. Since the semantics are approximated this is a less precise method than the exact, though bounded, semantic models used in our implementation of Dyno.

Chapter 8

Conclusion

In this thesis, we have explored behavioral pattern matching of programs written in an imperative-style toy language Imp. To this end, we defined and implemented a small pattern language called Dyno. Using an mCRL2 specification we extract a behavioral model from Imp programs. Then, after compiling Dyno patterns to μ-calculus formulas, we leverage mCRL2 to perform pattern matching by model checking these formulas against models extracted from Imp programs.

The translations of Dyno to μ-formulas, and Imp programs to mCRL2 specifications, were automated as transformations in Stratego. Using a set of automated tests, we verified the correctness of our implementation. Additionally, we surveyed the expressiveness of Dyno by discussing examples of expressible patterns, including some that implement common static analyses.

Some holes in the expressiveness of Dyno are identified, for which we propose new pattern constructs to be implemented in the future. Most notable are the reflexive transitive closure pattern and scoped patterns. Another limitation we recognize is the absence of a termination guarantee when pattern matching programs with infinite state spaces. While we provide methods for limiting state spaces by bounding unknown input parameter values and return values of external functions, additional bounding methods are needed to guarantee termination for all programs.

Currently, the result of matching a pattern against a program is simply `true` or `false`, indicating whether the specified pattern occurs in the program. For pattern matching to be of real practical use, additional information has to be reported, such as bindings of metavariables. We leave this as an opportunity for future work.

Bibliography

Blackburn, Patrick and Johan van Benthem (2007). "Modal logic: a semantic perspective". In: *Handbook of Modal Logic*. Ed. by Patrick Blackburn, J. F. A. K. van Benthem, and Frank Wolter. Vol. 3. Studies in logic and practical reasoning. North-Holland, pp. 1–84. ISBN: 978-0-444-51690-9. DOI: 10.1016/s1570-2464(07)80004-8. URL: https://doi.org/10.1016/s1570-2464(07)80004-8.

Brunel, Julien et al. (2009). "A foundation for flow-based program matching: using temporal logic and model checking". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, pp. 114–126. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480897. URL: http://doi.acm.org/10.1145/1480881.1480897.

Bunte, Olav et al. (2019). "The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11428. Lecture Notes in Computer Science. Springer, pp. 21–39. ISBN: 978-3-030-17465-1. DOI: 10.1007/978-3-030-17465-1_2. URL: https://doi.org/10.1007/978-3-030-17465-1_2.

Cousot, Patrick and Radhia Cousot (1976). "Static determination of dynamic properties of programs". English (US). In: *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, pp. 106–130.

Groote, Jan Friso and Mohammad Reza Mousavi (2014). *Modeling and Analysis of Communicating Systems*. MIT Press. ISBN: 9780262321020. URL: https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems.

Hennessy, Matthew and Robin Milner (1980). "On Observing Nondeterminism and Concurrency". In: *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*. Ed. by J. W. de Bakker and Jan van Leeuwen. Vol. 85. Lecture Notes in Computer Science. Springer, pp. 299–309. ISBN: 3-540-10003-2.

Kats, Lennart C. L. and Eelco Visser (2010). "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497. URL: https://doi.org/10.1145/1869459.1869497.

Kozen, Dexter (1983). "Results on the Propositional mu-Calculus". In: *Theoretical Computer Science* 27, pp. 333–354.

Lawall, Julia and Gilles Muller (2018). "Coccinelle: 10 Years of Automated Evolution in the Linux Kernel". In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston,*

MA, USA, July 11-13, 2018. Ed. by Haryadi S. Gunawi and Benjamin Reed. USENIX Association, pp. 601–614. URL: https://www.usenix.org/conference/atc18/presentation/lawall.

Miljak, Luka, Casper Bach Poulsen, and Rosilde Corvino (2024). "Concrete Syntax Metapatterns". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*. SLE '24. Pasadena, CA, USA: Association for Computing Machinery, pp. 43–55. ISBN: 9798400711800. DOI: 10.1145/3687997.3695637. URL: https://doi.org/10.1145/3687997.3695637.

Rice, H. G. (1953). "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Trans. Amer. Math. Soc.* 74, pp. 358–366.

Smits, Jeff and Eelco Visser (2020). "Gradually typing strategies". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*. Ed. by Ralf Lämmel, Laurence Tratt, and Juan de Lara. ACM, pp. 1–15. ISBN: 978-1-4503-8176-5. DOI: 10.1145/3426425.3426928. URL: https://doi.org/10.1145/3426425.3426928.

Spaendonck, P. (Nov. 2024). *Semi-Automatic Extraction of Formal Models from Object Oriented Code*. DOI: 10.48550/arXiv.2411.12386.

Acronyms

AST abstract syntax tree

DSL domain-specific language

HML Hennessy-Milner logic

LTS labeled transition system

SmPL semantic patching language

CTL computation tree logic

CTL-VW computation tree Logic with variables and witnesses

LPS linear process specification

Appendix A

mCRL2 code

This appendix contains the full mCRL2 boilerplate for transforming Imp programs into LTSs that represent their state space.

```
% %%%%%%%%%%%%%%%%%%%%%%% %
% LANGUAGE DEFINITIONS %
% %%%%%%%%%%%%%%%%%%%%%%% %

sort Val = struct BoolV(get_bool: Bool) ? is_bool
                | IntV(get_int: Int) ? is_int
                | VoidV ? is_void
                ;

sort Stmt = struct assign(get_var: Id, get_expr: Expr) ? is_assign
                | glob_assign(get_var: Id, get_expr: Expr) ? is_glob_assign
                | ite(get_cond: Expr, get_then: Stmt, get_else: Stmt) ? is_ite
                | expr(get_expr: Expr) ? is_expr
                | return(get_expr: Expr) ? is_return
                | while(get_cond: Expr, get_body: Stmt) ? is_while
                | call(get_var: Id, get_fun: Id,
                       get_args: List(Expr)) ? is_call
                | blk(get_stmts: List(Stmt)) ? is_blk

                % auxiliary
                | while'(get_cond: Expr, get_body: Stmt) ? is_while'
                | call'(get_var: Id, get_fun: Id,
                       get_args: List(Expr)) ? is_call'
                | return'(get_expr: Expr) ? is_return'
                | assign'(get_var: Id, get_expr: Expr) ? is_assign'
                | glob_assign'(get_var: Id, get_expr: Expr) ? is_glob_assign'
                | end_stmt'(get_stmt: Stmt) ? is_end_stmt'
                ;

sort Expr = struct constant(get_val: Val) ? is_constant
                | variable(get_id: Id) ? is_variable
                | glob_variable(get_id: Id) ? is_glob_variable
                | unop(get_unop: UnOp, get_expr: Expr) ? is_unop
                | binop(get_binop: BinOp, get_left: Expr,
                       get_right: Expr) ? is_binop

```

```
37                 % auxiliary
38                 | end_expr'(get_expr: Expr) ? is_end_expr'
39                 ;
40
41 sort UnOp  = struct Not | Neg;
42 sort BinOp = struct And | Or | Eq | Neq | Add | Sub | Lt | Lte | Gt | Gte;
43
44 % %%%%%%%%%%% %
45 % INTERPRETER %
46 % %%%%%%%%%%% %
47 sort Env   = Id -> Val;
48
49 map empty_env: Env;
50 eqn empty_env = lambda id: Id. VoidV;
51
52 % Helper function for instantiating environments
53 map make_env: List(Id) # List(Val) -> Env;
54 var v: Val;
55     vs: List(Val);
56     id: Id;
57     ids: List(Id);
58 eqn make_env([],[]) = empty_env;
59     make_env(id|>ids,v|>vs) = make_env(ids,vs)[id->v];
60
61 map eval: Expr # Env # Env -> Val;
62 var uop: UnOp;
63     bop: BinOp;
64     e, e': Expr;
65     es: List(Expr);
66     v: Val;
67     env: Env;
68     genv: Env;
69     id: Id;
70 eqn eval(constant(v),       env, genv) = v;
71     eval(variable(id),      env, genv) = env(id);
72     eval(glob_variable(id), env, genv) = genv(id);
73     eval(unop(uop, e),      env, genv) = eval_unop(uop, eval(e, env, genv));
74     eval(binop(bop, e, e'), env, genv) = eval_binop(bop, evals([e, e'], env,
75                                                        genv));
76
77 map eval_unop: UnOp # Val -> Val;
78 var b: Bool;
79     i: Int;
80     env: Env;
81     uop: UnOp;
82 eqn eval_unop(Not, BoolV(b)) = BoolV(!b);
83     eval_unop(Neg, IntV(i))  = IntV(-i);
84
85 map eval_binop: BinOp # List(Val) -> Val;
86 var b, b': Bool;
87     i, i': Int;
88     v, v': Val;
```

```
    env: Env;
    bop: BinOp;
eqn eval_binop(And, [BoolV(b), BoolV(b')]) = BoolV(b && b');
    eval_binop(Or,  [BoolV(b), BoolV(b')]) = BoolV(b || b');
    eval_binop(Eq,  [v, v'])               = BoolV(v == v');
    eval_binop(Neq, [v, v'])               = BoolV(v != v');
    eval_binop(Add, [IntV(i), IntV(i')])   = IntV(i + i');
    eval_binop(Sub, [IntV(i), IntV(i')])   = IntV(i - i');
    eval_binop(Lt,  [IntV(i), IntV(i')])   = BoolV(i < i');
    eval_binop(Lte, [IntV(i), IntV(i')])   = BoolV(i <= i');
    eval_binop(Gt,  [IntV(i), IntV(i')])   = BoolV(i > i');
    eval_binop(Gte, [IntV(i), IntV(i')])   = BoolV(i >= i');

map evals: List(Expr) # Env # Env -> List(Val);
var e: Expr;
    es: List(Expr);
    env: Env;
    genv: Env;
eqn evals([],      env, genv) = [];
    evals(e |> es, env, genv) = eval(e, env, genv) |> evals(es, env, genv);

% %%%%%%%%%%%%%%%%%%%% %
% PROCESS DEFINITION %
% %%%%%%%%%%%%%%%%%%%% %

act call: Id # List(Val) # Int;
    ret: Id # Val # Int;
    start_stmt, end_stmt: Stmt # Int;
    start_expr: Expr;
    end_expr: Expr # Val;

sort Frame = struct F(get_pid: Id, get_body: List(Stmt), get_env: Env,
                      get_ret_var: Id, get_end_stmts: List(Stmt));

proc P_call(pid: Id, body: Stmt, env: Env, genv: Env) =
  P_frame(pid, pid, [body], env, dummy, genv, [], []);

proc P_return(pid: Id, cid: Id, stack: List(Frame), genv: Env, ret_var: Id,
              ret_val: Val, end_stmts: List(Stmt)) =
    (#end_stmts > 0)
      -> end_stmt(head(end_stmts),#stack)
      . P_return(pid, cid, stack, genv, ret_var, ret_val, tail(end_stmts))
      <> ret(cid,ret_val,#stack-1)
      . (#stack > 0)
          -> P_frame(pid,
                     get_pid(head(stack)),
                     get_body(head(stack)),
                     get_env(head(stack))[ret_var->ret_val],
                     get_ret_var(head(stack)),
                     genv,
                     tail(stack),
                     get_end_stmts(head(stack)))
```

```
141           <> Stable(genv);
142
143 proc P_frame(pid: Id, cid: Id, body: List(Stmt), env: Env, ret_var: Id,
144             genv: Env, stack: List(Frame), end_stmts: List(Stmt)) =
145   (#body == 0) ->
146     P_return(pid, cid, stack, genv, ret_var, VoidV, end_stmts) <> (
147     is_return(head(body))      -> start_stmt(head(body),#stack)
148                                   . P_expr([get_expr(head(body))],
149                                            pid,cid,
150                                            [return'(get_expr(head(body)))]
151                                              ++ tail(body),
152                                            env,ret_var,genv,stack,
153                                            head(body) |> end_stmts)
154   + is_return'(head(body))     -> P_return(pid,cid,stack,genv,ret_var,
155                                              eval(get_expr(head(body)),env,genv),
156                                              end_stmts)
157   + is_expr(head(body))        -> start_stmt(head(body),#stack)
158                                   . P_expr([get_expr(head(body))],
159                                          pid,cid,
160                                          end_stmt'(head(body)) |> tail(body),
161                                          env,ret_var,genv,stack,
162                                          head(body) |> end_stmts)
163   + is_assign(head(body))      -> start_stmt(head(body),#stack)
164                                   . P_expr([get_expr(head(body))],
165                                          pid,cid,
166                                          assign'(get_var(head(body)),
167                                                  get_expr(head(body)))
168                                            |> tail(body),
169                                          env,ret_var,genv,stack,end_stmts)
170   + is_assign'(head(body))     -> end_stmt(assign(get_var(head(body)),
171                                                   get_expr(head(body))),
172                                            #stack)
173                                   . P_frame(pid,cid,
174                                            tail(body),
175                                            env[
176                                              get_var(head(body)) ->
177                                              eval(get_expr(head(body)),
178                                                  env,genv)
179                                            ],
180                                            ret_var,genv,stack,end_stmts)
181   + is_glob_assign(head(body)) -> start_stmt(head(body),#stack)
182                                   . P_expr([get_expr(head(body))],
183                                          pid,cid,
184                                          glob_assign'(get_var(head(body)),
185                                                       get_expr(head(body)))
186                                            |> tail(body),
187                                          env,ret_var,genv,stack,end_stmts)
188   + is_glob_assign'(head(body)) -> end_stmt(glob_assign(get_var(head(body)),
189                                                         get_expr(head(body))),
190                                            #stack)
191                                   . P_frame(pid,cid,
192                                            tail(body),
```

```
193                                         env,ret_var,
194                                         genv[
195                                           get_var(head(body))->
196                                           eval(get_expr(head(body)),
197                                                 env,genv)
198                                         ],
199                                         stack,end_stmts)
200   + is_blk(head(body)) -> start_stmt(head(body),#stack)
201                           . P_frame(pid,cid,
202                                     get_stmts(head(body))
203                                       ++ [end_stmt'(head(body))]
204                                       ++ tail(body),
205                                     env,ret_var,genv,stack,
206                                     head(body) |> end_stmts)
207   + is_ite(head(body)) ->
208       (eval(get_cond(head(body)),env,genv) == BoolV(true)) ->
209                                       start_stmt(head(body),#stack)
210                                     . P_expr([get_cond(head(body))],
211                                              pid,cid,
212                                              [get_then(head(body)),
213                                              end_stmt'(head(body))]
214                                                ++ tail(body),
215                                              env,ret_var,genv,stack,
216                                              head(body) |> end_stmts)
217                                     <> start_stmt(head(body),#stack)
218                                     . P_expr([get_cond(head(body))],
219                                              pid,cid,
220                                              [get_else(head(body)),
221                                              end_stmt'(head(body))]
222                                                ++ tail(body),
223                                              env,ret_var,genv,stack,
224                                              head(body) |> end_stmts)
225   + is_while(head(body))       ->
226       (eval(get_cond(head(body)),env,genv) == BoolV(true)) ->
227                                       start_stmt(head(body),#stack)
228                                     . P_expr([get_cond(head(body))],
229                                              pid,cid,
230                                              [get_body(head(body)),
231                                              while'(get_cond(head(body)),
232                                                      get_body(head(body))),
233                                              end_stmt'(head(body))]
234                                                ++ tail(body),
235                                              env,ret_var,genv,stack,
236                                              head(body) |> end_stmts)
237                                     <> start_stmt(head(body),#stack)
238                                     . P_expr([get_cond(head(body))],
239                                              pid,cid,
240                                              end_stmt'(head(body))
241                                                |> tail(body),
242                                              env,ret_var,genv,stack,
243                                              head(body) |> end_stmts)
244   + is_while'(head(body)) ->
```

```
245       (eval(get_cond(head(body)),env,genv) == BoolV(true)) ->
246                                     tau
247                                   . P_expr([get_cond(head(body))],
248                                            pid,cid,
249                                            [get_body(head(body)),
250                                             head(body)]
251                                              ++ tail(body),
252                                            env,ret_var,genv,stack,end_stmts)
253                                   <> tau
254                                   . P_expr([get_cond(head(body))],
255                                            pid,cid,tail(body),
256                                            env,ret_var,genv,stack,end_stmts)
257   + (is_call(head(body))) -> start_stmt(head(body),#stack)
258                              . P_expr(get_args(head(body)),
259                                       pid,cid,
260                                       [call'(get_var(head(body)),
261                                              get_fun(head(body)),
262                                              get_args(head(body))),
263                                        end_stmt'(head(body))]
264                                         ++ tail(body),
265                                       env,ret_var,genv,stack,
266                                       head(body) |> end_stmts)
267   + (is_call'(head(body)) && !is_external(get_fun(head(body))))
268                                    -> call(get_fun(head(body)),
269                                            evals(get_args(head(body)),env,genv),
270                                            #stack)
271                                     . P_frame(pid,
272                                               get_fun(head(body)),
273                                               [func_body(get_fun(head(body)))],
274                                               make_env(
275                                                 func_args(get_fun(head(body))),
276                                                 evals(get_args(head(body)),
277                                                       env,genv)),
278                                               get_var(head(body)),
279                                               genv,
280                                               F(cid,tail(body),env,ret_var,
281                                                 end_stmts) |> stack,
282                                               end_stmts)
283   + (is_call'(head(body)) && is_external(get_fun(head(body))))
284                                  -> call(get_fun(head(body)),
285                                          evals(get_args(head(body)),env,genv),0)
286                                   . sum v: Val .
287                                       bound(get_fun(head(body)),dummy,v)
288                                         -> ret(get_fun(head(body)),v,0)
289                                          . P_frame(pid,cid,tail(body),
290                                                    env[get_var(head(body))->v],
291                                                    ret_var,genv,stack,end_stmts)
292   + is_end_stmt'(head(body)) -> end_stmt(get_stmt(head(body)),#stack)
293                              . P_frame(pid,cid,tail(body),env,ret_var,genv,
294                                        stack,tail(end_stmts))
295   );
296
```

```
proc P_expr(exprs: List(Expr), pid: Id, cid: Id, body: List(Stmt), env: Env,
            ret_var: Id, genv: Env, stack: List(Frame),
            end_stmts: List(Stmt)) =
  (#exprs == 0) -> P_frame(pid,cid,body,env,ret_var,genv,stack,end_stmts) <>
  (
    (is_constant(head(exprs))
    || is_variable(head(exprs))
    || is_glob_variable(head(exprs)))
                               -> start_expr(head(exprs))
                                . end_expr(head(exprs),
                                          eval(head(exprs),env,genv))
                                . P_expr(tail(exprs),pid,cid,body,env,ret_var,
                                         genv,stack,end_stmts)
  + is_unop(head(exprs))       -> start_expr(head(exprs))
                               . P_expr([get_expr(head(exprs)),
                                        end_expr'(head(exprs))]
                                          ++ tail(exprs),
                                        pid,cid,body,env,ret_var,genv,stack,
                                        end_stmts)
  + is_binop(head(exprs))      -> start_expr(head(exprs))
                               . P_expr([get_left(head(exprs)),
                                        get_right(head(exprs)),
                                        end_expr'(head(exprs))]
                                          ++ tail(exprs),
                                        pid,cid,body,env,ret_var,genv,stack,
                                        end_stmts)
  + is_end_expr'(head(exprs)) -> end_expr(get_expr(head(exprs)),
                                          eval(get_expr(head(exprs)),env,genv))
                               . P_expr(tail(exprs),pid,cid,body,env,ret_var,
                                        genv,stack,end_stmts)
  );

  proc S(pid: Id, genv: Env) = sum vals: List(Val) .
                                 (is_within_bounds(pid, vals)) ->
                                     call(pid,vals,-1)
                                   . P_call(pid,func_body(pid),
                                          make_env(func_args(pid),vals),genv);

% %%%%%%%%%%%%%%%%%%%%%%%%%%% %
% BOUNDS HELPER FUNCTIONS %
% %%%%%%%%%%%%%%%%%%%%%%%%%%% %
map meets_bounds: Id # List(Id) # List(Val) -> Bool;
var pid, id: Id;
    ids: List(Id);
    v: Val;
    vs: List(Val);
eqn meets_bounds(pid, [],[]) = true;
    meets_bounds(pid, id|>ids, v|>vs) = bound(pid,id,v)
                                        && meets_bounds(pid,ids,vs);

map is_within_bounds: Id # List(Val) -> Bool;
var pid: Id;
```

```
349     vals: List(Val);
350 eqn is_within_bounds(pid, vals) = #vals == #func_args(pid)
351                                   && meets_bounds(pid,func_args(pid),vals);
352
353 % %%%%%%%%%%%% %
354 % INSTANTIATION %
355 % %%%%%%%%%%%% %
356 sort Id = struct dummy | ...;
357
358 map func_args: Id -> List(Id);
359 eqn ...
360
361 map func_body: Id -> Stmt;
362 eqn ...
363
364 map is_external: Id -> Bool;
365 eqn ...
366
367 map bound: Id # Id # Val -> Bool;
368 var v: Val;
369 eqn ...
370
371 map glob_init: Env;
372 eqn glob_init = empty_env[...->...]
373
374 proc Stable(genv: Env) = S(...)+...+S(...);
375
376 init Stable(glob_init);
```