Automatic Verification of Programs with Indirection

MSc Thesis *(Afstudeerscriptie)*

written by

Kyndylan Nienhuis
(born February 10th, 1989 in Amsterdam, The Netherlands)

under the supervision of **Prof Dr Jan van Eijck**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
September 4th, 2012	Prof Dr Jan van Eijck
	Dr Peter van Emde Boas
	Prof Dr Benedikt Löwe
	Dr Alban Ponse

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

Abstract

In the first part we prove the correctness of an existing verification algorithm, namely counterexample-driven abstraction refinement. To be able to state the correctness of the algorithm, we modify it such that it verifies programs that have a formal semantics. We use propositional dynamic logic and we give a denotational semantics and an equivalent structural operational semantics.

Then we consider a deterministic fragment of propositional dynamic logic. We improve the efficiency of the algorithm by exploiting determinism when present and we prove that this algorithm terminates on incorrect deterministic programs. Note that the algorithm will not always terminate on correct deterministic programs, since verification is undecidable in general.

Finally, we consider programs with indirection and we show that the introduced algorithms verify these programs inefficiently. We propose symbolic execution as an alternative way of computing path constraints to circumvent this inefficiency. Furthermore, the variant of symbolic execution we define removes indirection from symbolic terms which enables us to use a theorem prover that does not handle indirection.

Acknowledgements

I thank Jan van Eijck for supervising me. I could always drop by his office and our discussions were very valuable. Especially his idea to look at propositional dynamic logic was valuable and led me to change the framework of my thesis. I also thank my friend Hraban Luyat for helping me to start writing everything down. Finally, I thank the members of my thesis committee for their useful comments.

Contents

Chapter 1

Introduction

A specification of a computer program is a formal description of the properties that we want the program to have. Verifying a program means checking whether the program satisfies the properties in its specification.

We start with an example. Consider the program in Listing 1.1 on the following page. This program waits until a message is available and then processes the message. Other programs can also manipulate the messages, therefore there is a lock that a program has to hold when accessing or changing the messages. The program in Listing 1.1 acquires a lock while checking for a message, retrieving the message and processing the message. If there is no message available it releases the lock to allow other programs to make a message available.

We want that the program uses the lock in a correct way. Constructing a specification that precisely describes what we think is the correct way is not in the scope of this thesis. For this program we take the following property as its specification which captures at least a part of what it means to use the lock in a correct way.

> "The program does not call `lock` when it has the lock, or `unlock` when it does not have the lock."

To see that the example program satisfies this specification, observe that at the end of the loop we have `has_messages = true` if and only if the program has the lock.

1.1 Automatic verification

During an execution of a program it can be automatically checked whether this execution violates or respects the properties in the specification. This means that in principle a program can be verified by checking all possible executions. However, the number of possibilities is usually astronomically large so this method is seldom feasible. There are several ways to test a program as good as possible in a certain amount of time. One is selecting random executions and test the program using these states. See [3] for an example about random testing in Haskell. Another is directed testing. In this method constraints are constructed that lead the execution to a certain part of the program when satisfied by the initial state. See [15] for an example of directed testing of C

Listing 1.1 A program that calls `lock` and `unlock` in alternation

```
//Acquire message
has_messages = false;
while (not has_messages) {
    lock();
    has_messages = check_messages();
    if (has_messages) {
        message = get_message();
    }
    else {
        unlock();
        sleep();
    }
}
//Process message
do_stuff(message);
⋮
do_stuff(message);
unlock();
```

programs. Both methods have successfully been used to find bugs, but they can only prove a program correct by exhaustively checking all states.

Cousot and Cousot [6] introduced abstraction techniques that can be used to check multiple states at once. When essential details are abstracted away, verifying an abstraction has the possibility to lead to wrong results. To keep the error one-sided Clarke, Grumberg and Long [5] defined overapproximations that do not allow false positives. In [4] Clarke et al. showed how false negatives can be used to refine an overapproximation. This is called counterexample-driven refinement.

Counterexample-driven refinement has been implemented in the tool Yogi [1] that has been used to efficiently verify windows device drivers. In this algorithm the initial abstraction abstracts away from everything and only details are included that are necessary to avoid false negatives.

1.2 Overview

We restrict ourselves to partial correctness specifications. A partial correctness specification is a triple $\varphi_{pre}\,\{\alpha\}\,\varphi_{post}$ where α is a formal program and φ_{pre} and φ_{post} conditions on the program. It states that whenever α is executed in a state where φ_{pre} holds, then φ_{post} holds in every state that could be the result of executing α. This form of specifying was proposed by Floyd and Hoare [8, 10].

In Chapter 2 we introduce the framework we use to describe programs and specifications. We have chosen propositional dynamic logic (PDL) because it can express partial correctness specifications in a very simple and elegant way. We translate the program in 1.1 and its specification to PDL.

To verify a program we need a more detailed description of its execution than PDL gives. With this purpose in mind we define in Chapter 3 control flow graphs and we connect this with propositional dynamic logic.

Then in Chapter 4 we define the abstraction refinement algorithm that can verify propositional dynamic logic programs. We use overapproximations and counterexample-driven refinement as described before and we prove the correctness of this algorithm.

In Chapter 5 we consider a fragment of PDL that has deterministic code constructs and allows for deterministic and non-deterministic actions. We adapt the abstraction refinement algorithm such that it exploits the fact that some actions are deterministic. We prove that when all actions are deterministic, the algorithm will eventually terminate on programs that are incorrect. Note that this result cannot be extended to all correct or incorrect, deterministic programs, since verification is undecidable. We conclude this chapter by running the abstraction refinement algorithm on the PDL version of the program given in Listing 1.1.

Finally, in Chapter 6 we define a language that allows for multiple levels of indirection, similar to C with pointer arithmetic, and we motivate why indirection makes it infeasible to construct path constraints using weakest preconditions. We define a variant of symbolic execution that does not have a fixed mapping between symbols and input variables, but adds mappings when needed during the execution. This enables us to cope with the multiple levels of indirection and we show that it can be used as an alternative to constructing path constraints.

Chapter 2

Propositional dynamic logic

Propositional dynamic logic was developed by Pratt [14] and Fischer and Ladner [7]. In this thesis we will use the definition of Fischer and Ladner. PDL has a complete axiomatization, see for example [13].

For an introduction to propositional dynamic logic and an overview of its uses we recommend [16]. PDL is a multimodal logic, see [2] for a thorough introduction to modal logic.

We will define the language of propositional dynamic logic in Section 2.1, its semantics in Section 2.2 and in Section 2.3 we define a model that is able to capture the essence of the program described in Listing 1.1 on page 3.

2.1 Language

The language of propositional dynamic logic consists of formulas and programs. It is parametrized by a signature to be able to describe various other languages.

Definition 2.1 (Signature)**.** A signature is a pair $(\mathcal{P}, \mathcal{B})$ with $\mathcal{P}$ a set of propositions and $\mathcal{B}$ a set of basic actions.

Definition 2.2 (Formulas and programs)**.** With simultaneous recursion we define the set Φ of formulas and the set Π of programs over the signature $(\mathcal{P}, \mathcal{B})$. Let $p \in \mathcal{P}$, $b \in \mathcal{B}$, $\varphi_1, \varphi_2 \in \Phi$ and $\alpha_1, \alpha_2 \in \Pi$, then

$$\begin{array}{rcl} \varphi & ::= & \top \mid p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \langle\alpha_1\rangle\varphi_1 \\ \alpha & ::= & b \mid ?\varphi_1 \mid \alpha_1;\alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha_1^*. \end{array}$$

The intended meaning of $\langle\alpha\rangle\varphi$ is that there exists a computation of α that results in a state where φ is the case. The connection between propositional dynamic logic and modal logic is that $\langle\alpha\rangle$ can be seen as a modal operator.

The base cases in the definition of programs are programs of the form b or $?\varphi$. The former is a basic action and the latter a test action, that tests whether φ holds.

Programs can be composed in three ways: $\alpha_1;\alpha_2$ is the sequential composition of α_1 and α_2, $\alpha_1 \cup \alpha_2$ is the program that non-deterministically chooses between α_1 and α_2 and α^* is the program that executes α an arbitrary number $n \in \mathbb{Z}_{\geq 0}$ of times.

We use the following abbreviations

$$\begin{aligned}
\bot &= \neg\top \\
\varphi \lor \psi &= \neg\left(\neg\varphi \land \neg\psi\right) \\
\varphi \to \psi &= \neg\varphi \lor \psi \\
\varphi \leftrightarrow \psi &= (\varphi \to \psi) \land (\psi \to \varphi) \\
[\alpha]\varphi &= \neg\langle\alpha\rangle\neg\varphi.
\end{aligned}$$

Given the intended meaning of $\langle\alpha\rangle$, we see that $[\alpha]\varphi = \neg\langle\alpha\rangle\neg\varphi$ means that in every state that is a result of the execution of α we have that φ holds.

The modal operator $\langle\alpha\rangle$ enables us to express a partial correctness specification $\varphi_{pre}\,\{\alpha\}\,\varphi_{post}$. We have no need to allow program modalities in the conditions φ_{pre} and φ_{post}, thus we use the following definition.

Definition 2.3 (Specifications)**.** A partial correctness specification of a program α is a pair $(\varphi_{pre}, \varphi_{post}) \in \Phi \times \Phi$ where φ_{pre} and φ_{post} do not contain program modalities. Furthermore, we require that every test action $?\chi$ in α also does not contain program modalities. The propositional dynamic logic formula that states that α is correct is $\varphi_{pre} \to [\alpha]\varphi_{post}$.

2.2 Semantics

We define the semantics over a labeled transition system where labels are actions $b \in \mathcal{B}$. This is a generalization of a Kripke structure that is used as a model for modal logic.

Definition 2.4 (Models)**.** A model over the signature $(\mathcal{P}, \mathcal{B})$ is a tuple $M = (\Sigma, \mathcal{V}, \mathcal{R})$ where Σ is a set of states, $\mathcal{V}$ a valuation that sends $p \in \mathcal{P}$ to the set $\mathcal{V}(p) \subseteq \Sigma$ where p is true and $\mathcal{R}$ a function from labels to transitions that sends $b \in \mathcal{B}$ to the binary relation $\mathcal{R}(b) \subseteq \Sigma \times \Sigma$.

The binary relation $\mathcal{R}(b)$ describes the meaning of the basic action b as follows. We say $(s, r) \in \mathcal{R}(b)$ when r is a possible resulting state of executing b in the state s. When there are multiple $r \in \Sigma$ with $(s, r) \in \mathcal{R}(b)$ then b is a non-deterministic action. When there is no $r \in \Sigma$ with $(s, r) \in \mathcal{R}(b)$ the action b cannot be executed in s.

The interpretation of a program α will be a binary relation with the same meaning. Before we can define this, we need the following general definitions about binary relations.

Definition 2.5. Let R_1 and R_2 be binary relations over a set S. The relational composition is defined as

$$R_1 \circ R_2 = \left\{(s_1, s_3) \in S \times S \mid \exists s_2 \in S\ (s_1, s_2) \in R_1 \land (s_2, s_3) \in R_2\right\}.$$

The n-fold composition of a relation R with itself is recursively defined

$$\begin{aligned}
R^0 &= \{(s, s) \mid s \in S\} \\
R^{n+1} &= R^n \circ R.
\end{aligned}$$

The reflexive transitive closure of R is given by

$$R^* = \bigcup_{i=0}^{\infty} R^i.$$

Definition 2.6 (Semantics)**.** The interpretation of a formula $\varphi \in \Phi$ in a model $M = (\Sigma, \mathcal{V}, \mathcal{R})$ is a set of states $\mathcal{I}^M(\varphi) \subseteq \Sigma$ where φ is true and the interpretation of a program $\alpha \in \Pi$ is a binary relation $\mathcal{I}^M(\alpha) \subseteq \Sigma \times \Sigma$. We define $\mathcal{I}^M$ by simultaneous recursion.

$$\begin{aligned}
\mathcal{I}^M(\top) &= \Sigma \\
\mathcal{I}^M(p) &= \mathcal{V}(p) \\
\mathcal{I}^M(\neg\varphi_1) &= \Sigma - \mathcal{I}^M(\varphi_1) \\
\mathcal{I}^M(\varphi_1 \wedge \varphi_2) &= \mathcal{I}^M(\varphi_1) \cap \mathcal{I}^M(\varphi_2) \\
\mathcal{I}^M(\langle\alpha_1\rangle\varphi_1) &= \left\{s \in \Sigma \mid \exists t \in \Sigma\ (s,t) \in \mathcal{I}^M(\alpha_1) \wedge t \in \mathcal{I}^M(\varphi_1)\right\}.
\end{aligned}$$

$$\begin{aligned}
\mathcal{I}^M(b) &= \mathcal{R}(b) \\
\mathcal{I}^M(?\varphi) &= \left\{(s,s) \in \Sigma \times \Sigma \mid s \in \mathcal{I}^M(\varphi)\right\} \\
\mathcal{I}^M(\alpha_1;\alpha_2) &= \mathcal{I}^M(\alpha_1) \circ \mathcal{I}^M(\alpha_2) \\
\mathcal{I}^M(\alpha_1 \cup \alpha_2) &= \mathcal{I}^M(\alpha_1) \cup \mathcal{I}^M(\alpha_2) \\
\mathcal{I}^M(\alpha_1^*) &= \left(\mathcal{I}^M(\alpha_1)\right)^*.
\end{aligned}$$

The binary relation $\mathcal{I}^M(\alpha)$ describes the program α by specifying every pair (s,r) of states such that r is a possible end state when α is executed in the state s.

We write $M, s \vDash \varphi$ for $s \in \mathcal{I}^M(\varphi)$. When M is clear from the context, we abbreviate $M, s \vDash \varphi$ to $s \vDash \varphi$. We say $M \vDash \varphi$ if for all $s \in \Sigma$ we have $s \vDash \varphi$.

Definition 2.7 (Verification)**.** Let $(\varphi_{pre}, \varphi_{post})$ be a specification of the program α. Verifying α against this specification is deciding whether $M \vDash \varphi_{pre} \to [\alpha]\varphi_{post}$ or $M \nvDash \varphi_{pre} \to [\alpha]\varphi_{post}$.

2.3 A non-deterministic boolean model

We define a model that serves as an example of the framework we defined in this chapter and that can be used to describe the program of Listing 1.1 on page 3. The language consists of boolean variables and deterministic and non-deterministic assignments. We will call this model NDB.

Let *bool* $= \{\top, \bot\}$. Define the signature $(\mathcal{P}, \mathcal{B})$ where $\mathcal{P}$ is a set of variables and

$$\mathcal{B} = \{p := \mathit{random} \mid p \in \mathcal{P}\} \cup \{p := x \mid p \in \mathcal{P}, x \in \mathit{bool}\}.$$

We take the model $M = (\Sigma, \mathcal{V}, \mathcal{R})$ where Σ is the set of functions from $\mathcal{P}$ to *bool*,

$$\begin{aligned}
\mathcal{V}(p) &= \{s \in \Sigma \mid s(p) = \top\}, \\
(s,r) &\in \mathcal{R}(p := \mathit{random}) \text{ iff } s(q) = r(q) \text{ for all } q \neq p, \\
(s,r) &\in \mathcal{R}(p := x) \text{ iff } r(p) = x \text{ and for all } q \neq p \text{ we have } s(q) = r(q).
\end{aligned}$$

Example 2.8. Consider the program defined in Listing 1.1 on page 3. We will describe this program in NDB. We use the three variables lock, $\mathit{has\_m}$ and foo. The first denotes whether the lock has been acquired or not, the second whether there is a message to process and the last to do irrelevant calculations with.

Every time the function `check_messages()` is called in the program, it can return `true` and `false` arbitrarily. Hence, we model it using a random assignment. We also model the irrelevant computations with random assignments, because we do not care about what happens. The translation of the statements is thus as follows

$$\begin{aligned}
\texttt{lock()} &\mapsto \mathit{lock} := \top \\
\texttt{unlock()} &\mapsto \mathit{lock} := \bot \\
\texttt{has\_messages = false} &\mapsto \mathit{has\_m} := \bot \\
\texttt{has\_messages = check\_messages()} &\mapsto \mathit{has\_m} := \mathit{random} \\
\texttt{message = get\_message()} &\mapsto foo := \mathit{random} \\
\texttt{sleep()} &\mapsto foo := \mathit{random} \\
\texttt{do\_stuff(message)} &\mapsto foo := \mathit{random}.
\end{aligned}$$

We model the control flow statements as follows, see Chapter 5 for the intuition behind this choice of modeling.

$$\begin{aligned}
\texttt{if } (\varphi)\{\alpha_1\} \texttt{ else } \{\alpha_2\} &\mapsto (?\varphi;\alpha_1) \cup (?\neg\varphi;\alpha_2) \\
\texttt{while } (\varphi)\ \{\alpha\} &\mapsto (?\varphi;\alpha)^*\,;?\neg\varphi.
\end{aligned}$$

The translated program is given in Figure 2.1 on the following page.

As stated in the introduction the specification we want to check is “The program does not call `lock` when it has the lock, or `unlock` when it does not have the lock.” To model this in our framework we introduce a new variable error. We replace the following calls

$$\begin{aligned}
\mathit{lock} := \top &\mapsto (?\mathit{lock};\mathit{error} := \top) \cup (?\neg\mathit{lock};\mathit{lock} := \top) \\
\mathit{lock} := \bot &\mapsto (?\mathit{lock};\mathit{lock} := \bot) \cup (?\neg\mathit{lock};\mathit{error} := \top)\,.
\end{aligned}$$

We require that error is false when the program starts, then we know that error will be true at the end of the execution if and only if the program violated the property. We assume that the program does not have the lock when it starts and we require that it has released the lock when the program has finished. This leads to the following specification of the program $\varphi_{pre} = \varphi_{post} = (\neg\mathit{error} \wedge \neg\mathit{lock})$. The full code is given in Figure 2.2.

Listing 2.1 Lock/unlock program in NDB

```
has_m := ⊥;
(
    ? ¬has_m;
    lock := ⊤;
    has_m := random;
    (
        ? has_m;
        foo := random
    ∪
        ? ¬has_m;
        lock := ⊥;
        foo := random
    )
)*;
? has_m;
foo := random;
lock := ⊥
```

Listing 2.2 Lock/unlock program with error check

```
has_m := ⊥;
(
    ? ¬has_m;
    (
        ? lock;
        error := ⊤
    ∪
        ? ¬lock;
        lock := ⊤
    );
    has_m := random;
    (
        ? has_m;
        foo := random
    ∪
        ? ¬has_m;
        (
            ? lock;
            lock := ⊥
        ∪
            ? ¬lock;
            error := ⊤
        );
        foo := random
    )
)*;
? has_m;
foo := random;
(
    ? lock;
    lock := ⊥
∪
    ? ¬lock;
    error := ⊤
)
```

Chapter 3

Control flow graphs

While propositional dynamic logic makes it easy to describe a program and its specification, the recursive definition of programs makes it difficult to prove or disprove that a program satisfies its specification. To overcome this we will introduce semantics that describes all the individual steps of executing a program. From now on, we will call the semantics $\mathcal{I}$ defined in the previous chapter the denotational semantics and use the notation $\mathcal{I}_{\text{ds}}$. The semantics defined in this chapter is called the structural operational semantics, it is denoted by $\mathcal{I}_{\text{sos}}$. See [12] for a comparison between different styles of defining semantics.

We will use control flow graphs to define the structural operational semantics. Control flow graphs have a language and models of its own, to avoid confusion we will call the signatures and models defined in the previous chapter PDL-signatures and PDL-models and the signature and models of control flow graphs CFG-signatures and CFG-models. In later chapters we will not be concerned with CFG-signatures and CFG-models anymore, and then we will drop the prefix PDL. The notation for the semantics of control flow graphs is $\mathcal{I}_{\text{graph}}$.

In Section 3.1 we define control flow graphs, CFG-models and the interpretation $\mathcal{I}_{\text{graph}}$. In the next section we will define a translation from programs to control flow graphs and use this to define the structural operational semantics $\mathcal{I}_{\text{sos}}$. Then in Section 3.3 we will prove that the structural operational and denotational semantics are equivalent.

3.1 Language and semantics

The language of control flow graphs is parametrized by a set of actions. When we make the connection with PDL, this set will differ from the set $\mathcal{B}$ in a PDL-signature $(\mathcal{P}, \mathcal{B})$. Hence, we define CFG-signatures separately.

Definition 3.1 (CFG-signature)**.** A CFG-signature is a set $\mathcal{A}$ of actions.

We will use a control flow graph to represent a program. It is defined as a finite, directed multigraph with its edges labeled by actions. A node represents a certain stage during the execution of the program. There is a special initial node I and a final node F that represent the starting and ending stage. Outgoing edges from a node specify which actions can be executed next at that stage. We define this formally as follows.

Definition 3.2 (Control flow graphs)**.** A control flow graph G over $\mathcal{A}$ is a tuple (N, I, F, E) where N is a finite set of nodes, $I, F \in N$ are its initial and final nodes and $E \subseteq N \times \mathcal{A} \times N$ a set of directed, labeled edges.

We will use the following notation for a path γ that traverses the nodes $g_0, \ldots, g_n$ along edges labeled with the actions $a_1, \ldots, a_n$

$$\gamma = g_0 \overset{a_1}{\to} g_1 \overset{a_2}{\to} \ldots \overset{a_n}{\to} g_n.$$

CFG-models have exactly the same structure as PDL-models, except that there is no need for a valuation function $\mathcal{V}$. Since $\mathcal{A}$ can differ from the set $\mathcal{B}$, we cannot simply use the relevant aspects of a PDL-model and we need to define CFG-models separately.

Definition 3.3 (CFG-models)**.** A CFG-model K over the signature $\mathcal{A}$ is a tuple $(\Sigma, \mathcal{R})$ where Σ is a set of states and $\mathcal{R}$ a function that maps $a \in \mathcal{A}$ to the binary relation $\mathcal{R}(a) \subseteq \Sigma \times \Sigma$.

We have said that nodes represent stages during a computation and outgoing edges the possible next actions. A path in a control flow graph therefore represents a part of the computation. We use the word trace as a synonym for a finite sequence of states. We define the connection between traces and paths in the control flow graph as follows.

Definition 3.4 (Satisfying traces)**.** Let γ be a path of length n in the control flow graph G, let $a_1, \ldots, a_n$ be the labels of the edges. We say that the trace $s_0, \ldots, s_n$ of length $n+1$ with $s_i \in \Sigma$ satisfies γ if for all $i \in \{1, \ldots, n\}$ we have $(s_{i-1}, s_i) \in \mathcal{R}(a_i)$.

The possible paths in G that start at the initial node I and end at the final node F define the program that G represents.

Definition 3.5 (Semantics)**.** The interpretation of a control flow graph G in the CFG-model K is a binary relation $\mathcal{I}^K_{\text{graph}}(G) \subseteq \Sigma \times \Sigma$. It is defined by $(s, r) \in \mathcal{I}^K_{\text{graph}}(G)$ iff there is a path γ in G that starts at I and ends at F and a trace $(s, t_1, \ldots, t_{n-1}, r)$ that satisfies γ.

3.2 Connection with propositional dynamic logic

We will use control flow graphs to represent PDL programs. We have defined control flow graphs over a set $\mathcal{A}$ of actions. Since programs can contain basic actions $b \in \mathcal{B}$ and test actions $?\varphi$, we translate a PDL-signature $(\mathcal{P}, \mathcal{B})$ therefore to the CFG-signature $\mathcal{A} = \mathcal{B} \cup \{?\varphi \mid \varphi \in \Phi\}$ that is the union of basic and test actions.

Before we define the translation from programs $\alpha \in \Pi$ to control flow graphs, we define how to merge two nodes in an arbitrary graph. See Figure 3.1 on the next page for an example.

Definition 3.6 (Merging nodes)**.** Let G be a graph with nodes N and edges $E \subseteq N \times \mathcal{A} \times N$ and let $g_0, g_1 \in N$ be the nodes that will be merged. We define the graph $G' = (N', E')$ that is the result of merging g_0 and g_1 as follows.

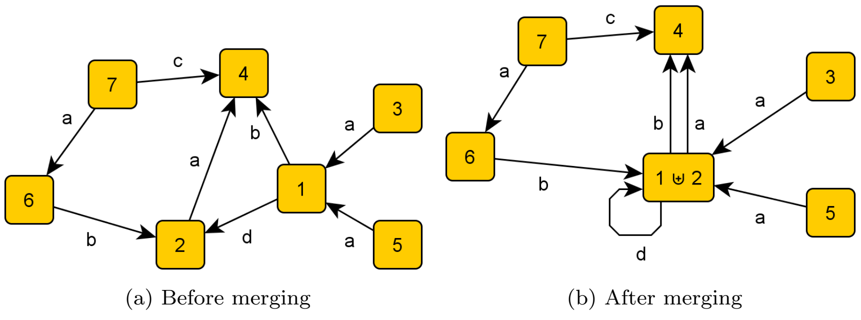


Figure 3.1: Merging the nodes 1 and 2

Let g be a new node, we write $g = g_0 \uplus g_1$. The set of nodes of G' is given by

$$N' = N \cup \{g\} \backslash \{g_0, g_1\}.$$

We do not want to change anything essential about the edges. So if g_0 or g_1 is part of an edge in G, then g will have the same role in an edge in G'. Formally, we define E' as follows

$$\begin{aligned} E' \;=\; & E \cap (N' \times \mathcal{A} \times N') \\ & \cup \{(h, a, g) \mid a \in \mathcal{A}, h \in N' \ (h, a, g_0) \in E \text{ or } (h, a, g_1) \in E\} \\ & \cup \{(g, a, h) \mid a \in \mathcal{A}, h \in N' \ (g_0, a, h) \in E \text{ or } (g_1, a, h) \in E\} \\ & \cup \{(g, a, g) \mid a \in \mathcal{A} \text{ if there are } x, y \in \{g_0, g_1\} \text{ with } (x, a, y) \in E\}. \end{aligned}$$

We can now define the translation from PDL programs to control flow graphs.

Definition 3.7. Let $\alpha \in \Pi$ be a program. We define its control flow graph $G(\alpha)$ with induction to the structure of α. See Figure 3.2 on the following page for an illustration.

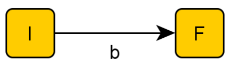
Case 1. Base cases: $\alpha = a$ or $\alpha = ?\varphi$. Define G as the graph with two nodes I and F; and an edge from I to F labeled with the (basic or test) action α.

Case 2. Recursive cases. Let G_1 and G_2 be the control flow graphs of α_1 and α_2 respectively and let I_1, I_2 and F_1, F_2 denote their respective initial and final nodes.

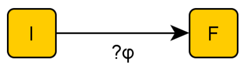
Case i. $\alpha = \alpha_1 ; \alpha_2$. First define G as the disjoint union of G_1 and G_2. Set $I = I_1$ and $F = F_2$ and merge the node F_1 with I_2.

Case ii. $\alpha = \alpha_1 \cup \alpha_2$. First define G as the disjoint union of G_1 and G_2. Set $I = I_1$ and $F = F_1$ and merge the node I_1 with I_2 and F_1 with F_2.

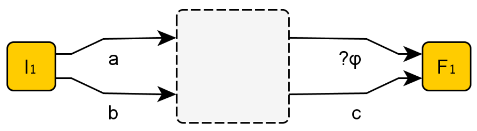
Case iii. $\alpha = \alpha_1^*$. Define G as the disjoint union of G_1 with two new nodes I and F. Add the following edges $I \to I_1$, $I \to F$, $F_1 \to I_1$ and $F_1 \to F$, all labeled with the action $?\top$.



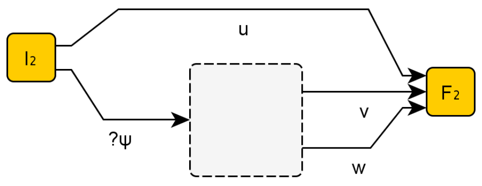
(a) The control flow graph of b



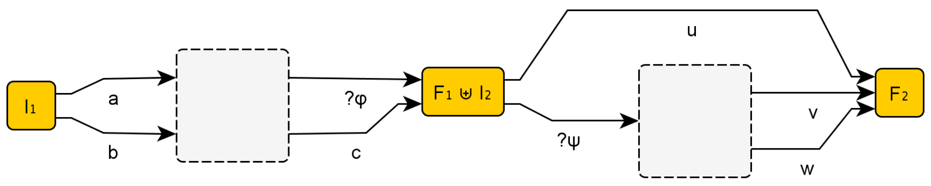
(b) The control flow graph of $?\varphi$



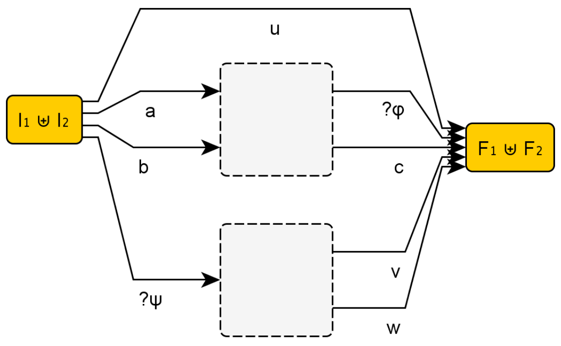
(c) The control flow graph of α_1



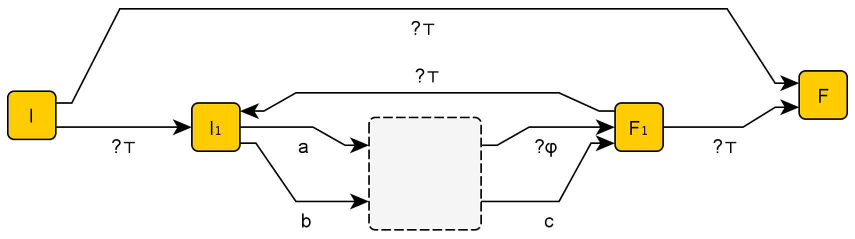
(d) The control flow graph of α_2



(e) The control flow graph of $\alpha_1; \alpha_2$



(f) The control flow graph of $\alpha_1 \cup \alpha_2$



(g) The control flow graph of α_1^*

Figure 3.2: Control flow graphs

Example 3.8. Recall the lock/unlock program defined in Listing 2.2 on page 10. In Figure 3.3 on the following page its control flow graph is shown.

From Figure 3.2 on the previous page we can make the following observation. We will need this detail when defining the verification algorithm.

Fact 3.9. *Let α be a program, there does not exist a node in the control flow graph $G(\alpha)$ that has an edge to itself.*

Now we are ready to define the structural operational semantics. Since programs can contain formulas as test actions and formulas can contain programs in modalities, the interpretation of formulas and the interpretation of programs depend on each other. We will let the structural operational interpretation $\mathcal{I}_{\text{sos}}$ of programs depend on the denotational interpretation $\mathcal{I}_{\text{ds}}$ of formulas. The denotational interpretation of formulas is however dependent on the denotational interpretation of programs, so indirectly the definition of $\mathcal{I}_{\text{sos}}$ of programs depends on the definition of $\mathcal{I}_{\text{ds}}$ of programs.

This apparent dependency disappears when we have proven that $\mathcal{I}_{\text{sos}}(\alpha) = \mathcal{I}_{\text{ds}}(\alpha)$. Then the structural operational interpretation of programs can be seen on its own right.

Definition 3.10 (Structural operational semantics)**.** Let $M = \left(\Sigma^M, \mathcal{V}^M, \mathcal{R}^M\right)$ be a PDL-model. The interpretation a program $\alpha \in \Pi$ is a binary relation $\mathcal{I}^M_{\text{sos}}(\alpha) \subseteq \Sigma^M \times \Sigma^M$. Define the CFG-model $K = \left(\Sigma^K, \mathcal{R}^K\right)$ where $\Sigma^K = \Sigma^M$ and

$$\mathcal{R}^K(a) = \begin{cases} \mathcal{R}^M(b) & \text{if } a = b \in \mathcal{B} \\ \left\{(s,s) \mid s \in \mathcal{I}^M_{\text{ds}}(\varphi)\right\} & \text{if } a = ?\varphi, \varphi \in \Phi. \end{cases}$$

Then define

$$\mathcal{I}^M_{\text{sos}}(\alpha) = \mathcal{I}^K_{\text{graph}}(G(\alpha)).$$

3.3 Equivalence of semantics

The proof that $\mathcal{I}^M_{\text{sos}}(\alpha) = \mathcal{I}^M_{\text{ds}}(\alpha)$ will be with induction to the structure of α. We will first prove the base case.

Lemma 3.11. *Let $M = \left(\Sigma^M, \mathcal{V}^M, \mathcal{R}^M\right)$ be a PDL-model. Let $a \in \mathcal{A}$ be a basic or test action, we have $\mathcal{I}^M_{\text{sos}}(a) = \mathcal{I}^M_{\text{ds}}(a)$.*

Proof. Let $K = \left(\Sigma^K, \mathcal{R}^K\right)$ be the CFG-model as defined in Definition 3.10. We have $\mathcal{I}^M_{\text{sos}}(a) = \mathcal{I}^K_{\text{graph}}(G(a))$. The control flow graph $G(a)$ is the graph $I \to F$ where the edge is labeled with a, see Definition 3.7 on page 13. Since there is only one path from I to F, we have $(s,r) \in \mathcal{I}^K_{\text{graph}}(G(a))$ if and only if $(s,r) \in \mathcal{R}^K(a)$.

Case 1. $a = b$ with $b \in \mathcal{B}$. By definition, $\mathcal{R}^K(b) = \mathcal{R}^M(b)$ and $\mathcal{I}^M_{\text{ds}}(b) = \mathcal{R}^M(b)$. Hence $\mathcal{I}^M_{\text{sos}}(b) = \mathcal{I}^M_{\text{ds}}(b)$.

Case 2. $a = ?\varphi$ with $\varphi \in \Phi$. By definition

$$\mathcal{R}^K(?\varphi) = \left\{(s,s) \in \Sigma \times \Sigma \mid s \in \mathcal{I}^M_{\text{ds}}(\varphi)\right\}$$

which is the same as the definition of $\mathcal{I}^M_{\text{ds}}(?\varphi)$. Hence, $\mathcal{I}^M_{\text{sos}}(?\varphi) = \mathcal{I}^M_{\text{ds}}(?\varphi)$.

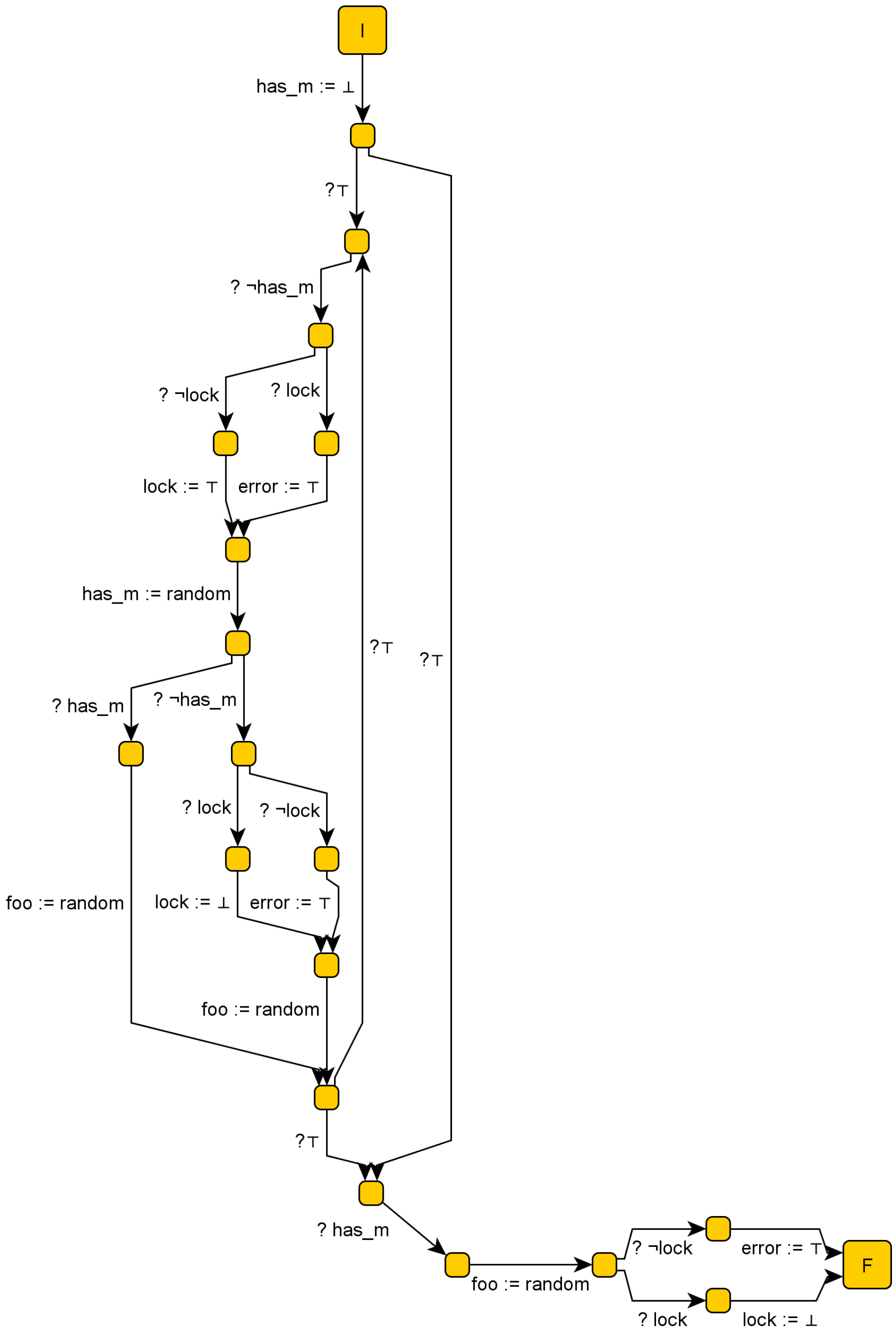


Figure 3.3: The control flow graph of the lock/unlock program

□

To clean up the notation, we now fix a model $M = (\Sigma, \mathcal{V}, \mathcal{R})$ and we omit the superscript M in $\mathcal{I}^M_{\text{ds}}$ and $\mathcal{I}^M_{\text{sos}}$.

Recall from Definition 2.6 on page 7 that the recursive cases of the definition of $\mathcal{I}_{\text{ds}}$ are given by

$$\begin{aligned}
\mathcal{I}_{\text{ds}}(\alpha_1;\alpha_2) &= \mathcal{I}_{\text{ds}}(\alpha_1) \circ \mathcal{I}_{\text{ds}}(\alpha_2)\\
\mathcal{I}_{\text{ds}}(\alpha_1 \cup \alpha_2) &= \mathcal{I}_{\text{ds}}(\alpha_1) \cup \mathcal{I}_{\text{ds}}(\alpha_2)\\
\mathcal{I}_{\text{ds}}(\alpha_1^*) &= (\mathcal{I}_{\text{ds}}(\alpha_1))^* .
\end{aligned}$$

For each of these cases, we will prove a lemma that states that $\mathcal{I}_{\text{sos}}$ behaves in the same way.

Lemma 3.12. *We have $\mathcal{I}_{\text{sos}}(\alpha_1;\alpha_2) = \mathcal{I}_{\text{sos}}(\alpha_1) \circ \mathcal{I}_{\text{sos}}(\alpha_2)$.*

Proof. The idea is that paths in $G(\alpha_1;\alpha_2)$ can be split into a path in $G(\alpha_1)$ and a path in $G(\alpha_2)$. Vice versa, a path γ_1 in $G(\alpha_1)$ and a path γ_2 in $G(\alpha_2)$ can be concatenated to form a path in $G(\alpha_1;\alpha_2)$, see Figure 3.2e on page 14.

Suppose $(s_1, s_3) \in \mathcal{I}_{\text{sos}}(\alpha_1;\alpha_2)$. Then there is a path γ in the control flow graph of $\alpha_1;\alpha_2$ and a trace σ from s_1 to s_3 that satisfies γ.

$$\begin{aligned}
\gamma &= I \overset{a_1}{\to} g_1 \overset{a_2}{\to} \dots \overset{a_{n-1}}{\to} g_{n-1} \overset{a_n}{\to} F\\
\sigma &= (t_0, \dots, t_n) \text{ with } t_0 = s_1, t_n = s_3.
\end{aligned}$$

This path has to cross $F_1 \uplus I_2$ somewhere, see Figure 3.2e on page 14, let i be the index such that $g_i = F_1 \uplus I_2$. Define the paths γ_1, γ_2 and the traces σ_1, σ_2

$$\begin{aligned}
\gamma_1 &= I_1 \overset{a_1}{\to} g_1 \overset{a_2}{\to} \dots \overset{a_{i-1}}{\to} g_{i-1} \overset{a_i}{\to} F_1\\
\gamma_2 &= I_2 \overset{a_{i+1}}{\to} g_{i+1} \overset{a_{i+2}}{\to} \dots \overset{a_{n-1}}{\to} g_{n-1} \overset{a_n}{\to} F_2\\
\sigma_1 &= (t_0, \dots, t_i)\\
\sigma_2 &= (t_i, \dots, t_n).
\end{aligned}$$

We have that γ_1 is a path in the control flow graph of α_1 and γ_2 in the control flow graph of α_2. Furthermore, σ_1 satisfies γ_1 and σ_2 satisfies γ_2. Hence, we have $(t_0, t_i) \in \mathcal{I}_{\text{sos}}(\alpha_1)$ and $(t_i, t_n) \in \mathcal{I}_{\text{sos}}(\alpha_2)$. We conclude that $(t_0, t_n) \in \mathcal{I}_{\text{sos}}(\alpha_1) \circ \mathcal{I}_{\text{sos}}(\alpha_2)$.

For the other direction, suppose $(s_1, s_3) \in \mathcal{I}_{\text{sos}}(\alpha_1) \circ \mathcal{I}_{\text{sos}}(\alpha_2)$. Then there is a s_2 with $(s_1, s_2) \in \mathcal{I}_{\text{sos}}(\alpha_1)$ and $(s_2, s_3) \in \mathcal{I}_{\text{sos}}(\alpha_2)$. Hence, there are paths γ_1 and γ_2 in the control flow graphs of respectively α_1 and α_2 with satisfying traces σ_1 and σ_2.

$$\begin{aligned}
\gamma_1 &= I_1 \overset{a_1}{\to} g_1 \overset{a_2}{\to} \dots \overset{a_{n-1}}{\to} g_{n-1} \overset{a_n}{\to} F_1\\
\gamma_2 &= I_2 \overset{b_1}{\to} h_1 \overset{b_2}{\to} \dots \overset{b_{k-1}}{\to} h_{k-1} \overset{b_k}{\to} F_2\\
\sigma_1 &= (r_0, \dots, r_n) \text{ with } r_0 = s_1, r_n = s_2\\
\sigma_2 &= (t_0, \dots, t_k) \text{ with } t_0 = s_2, t_k = s_3.
\end{aligned}$$

These paths can be concatenated to the path γ in the control graph of $\alpha_1;\alpha_2$ and these traces can be combined to σ as follows. Note that when combining σ_1 and σ_2 to σ we omitted t_0.

$$\begin{aligned}\gamma &= I \xrightarrow{a_1} g_1 \dots g_{n-1} \xrightarrow{a_n} F_1 \uplus I_2 \xrightarrow{b_1} h_1 \dots h_{k-1} \xrightarrow{b_k} F \\ \sigma &= (r_0, \dots, r_n, t_1 \dots, t_k).\end{aligned}$$

Since $r_n = t_0$ we have that σ satisfies γ. Since $r_0 = s_1$ and $t_k = s_3$ we have $(s_1, s_3) \in \mathcal{I}_{\text{sos}}(\alpha_1;\alpha_2)$. □

Lemma 3.13. *We have* $\mathcal{I}_{\text{sos}}(\alpha_1 \cup \alpha_2) = \mathcal{I}_{\text{sos}}(\alpha_1) \cup \mathcal{I}_{\text{sos}}(\alpha_2)$.

Proof. From Figure 3.2f on page 14 it follows that every path γ in the control flow graph $G(\alpha_1)$ is a path in the control flow graph of $\alpha_1 \cup \alpha_2$, the same holds for paths in $G(\alpha_2)$. Vice versa, a path γ in the control flow graph of $\alpha_1 \cup \alpha_2$ is a path in $G(\alpha_1)$ or in $G(\alpha_2)$ (or both). □

Lemma 3.14. *We have* $\mathcal{I}_{\text{sos}}(\alpha^*) = (\mathcal{I}_{\text{sos}}(\alpha))^*$.

Proof. The idea is that a path γ in $G(\alpha^*)$ is either $I \xrightarrow{?\top} F$ or can be split into n paths in $G(\alpha)$. In the other direction, n paths in $G(\alpha)$ can be concatenated to form a path in $G(\alpha^*)$. See Figure 3.2g on page 14.

Suppose $(s, r) \in \mathcal{I}_{\text{sos}}(\alpha^*)$. There is a path γ in the control flow graph of α^* and a trace σ that satisfies γ.

$$\begin{aligned}\gamma &= I \xrightarrow{a_1} g_1 \xrightarrow{a_2} \dots \xrightarrow{a_{m-1}} g_{m-1} \xrightarrow{a_m} F \\ \sigma &= (s_0, \dots, s_m) \text{ with } s_0 = s, s_m = r.\end{aligned}$$

Let $n \in \mathbb{Z}_{\geq 0}$ be the number of times F_1 occurs in γ, see Figure 3.2g. If $n = 0$, we must have $m = 1$ with $a_1 = ?\top$. Then $s = r$, so $(s, r) \in (\mathcal{I}_{\text{sos}}(\alpha))^0 \subseteq (\mathcal{I}_{\text{sos}}(\alpha))^*$.

Else, for $i = 1, \dots, n$ let k_i be the number such that $g_{k_i} = F_1$ and let $k_0 = 0$. Define the path γ_i as the part of γ starting at the $(k_{i-1}+1)$-th node and ending at the k_i-th node. We see that γ_i is a path in the control flow graph $G(\alpha)$ since $g_{k_{i-1}+1} = I_1$ and $g_{k_i} = F_1$. We split the trace σ in the same way in traces σ_i.

$$\begin{aligned}\gamma_i &= g_{k_{i-1}+1} \xrightarrow{a_{k_{i-1}+2}} \dots \xrightarrow{a_{k_i}} g_{k_i} \\ \sigma_i &= (s_{k_{i-1}+1}, \dots, s_{k_i})\end{aligned}$$

Since σ satisfies γ, we have that σ_i satisfies γ_i so $(s_{k_{i-1}+1}, s_{k_i}) \in \mathcal{I}_{\text{sos}}(\alpha)$. Since the only edge from F_1 to I_1 in the control flow graph of $\alpha^*$is labeled with $?\top$, and σ is a satisfying trace, we must have $s_{k_i} = s_{k_1+1}$. Hence, $(s, r) \in (\mathcal{I}_{\text{sos}}(\alpha))^n \subseteq (\mathcal{I}_{\text{sos}}(\alpha))^*$

For the other direction, let $(s, r) \in (\mathcal{I}_{\text{sos}}(\alpha))^*$. Then there is a $n \in \mathbb{Z}_{\geq 0}$ with $(s, r) \in (\mathcal{I}_{\text{sos}}(\alpha))^n$. If $n = 0$, we have $s = r$, so the trace (s, r) satisfies the path I, F with edge-label $?\top$ in the control flow graph of α^*. Hence, $(s, r) \in \mathcal{I}_{\text{sos}}(\alpha^*)$.

Else, for every $i = 1, \dots, n$ there is a path γ_i in the control flow graph of α and a satisfying trace $\sigma_i = (s_0^i, \dots, s_{m_i}^i)$ such that $s_{m_i}^i = s_0^{i+1}$ for $i < n$, $s_0^0 = s$ and $s_{m_n}^n = r$.

We construct the path γ as follows. It starts from I to I_1 using edge $?\top$. Then γ_1 is concatenated; this path ends in F_1. Then for all $i > 1$, we append the transition $?\top$ from F_1 to I_1 and we append the path γ_i. We see that this path also ends in F_1, so this construction is well-defined. Finally, we append the transition from F_1 to F using the edge $?\top$. From Figure 3.2g it follows that γ is a path of $G(\alpha^*)$.

Define the trace σ that is the concatenation of all traces σ_i

$$\sigma = (s_0^1, \ldots, s_{m_1}^1, s_0^2, \ldots s_{m_2}^2, \ldots, s_0^n, \ldots, s_{m_n}^n).$$

Since $s_{m_i}^i = s_0^{i+1}$ we have that $s_{m_i}^i, s_0^{i+1}$ satisfies the transition $?\top$. Then it follows from the fact that σ_i satisfies γ_i that σ satisfies γ. Hence, $(s, r) \in \mathcal{I}_{\text{sos}}(\alpha^*)$. □

We will now wrap everything up to prove the equivalence of the semantics.

Theorem 3.15. *Let M be a PDL-model. The structural operational interpretation $\mathcal{I}_{\text{sos}}^M(\alpha)$ equals the denotational interpretation $\mathcal{I}_{\text{ds}}^M(\alpha)$.*

Proof. For brevity, we will omit the superscript M in $\mathcal{I}_{\text{ds}}^M$ and $\mathcal{I}_{\text{sos}}^M$. We will prove this with induction to the structure of α.

Case 1. $\alpha = b$ and $\alpha = ?\varphi$. By Lemma 3.11.

Case 2. $\alpha = \alpha_1; \alpha_2$. we have

$$\begin{aligned} \mathcal{I}_{\text{sos}}(\alpha_1;\alpha_2) &= \mathcal{I}_{\text{sos}}(\alpha_1) \circ \mathcal{I}_{\text{sos}}(\alpha_2) \text{ by Lemma 3.12} \\ \mathcal{I}_{\text{ds}}(\alpha_1;\alpha_2) &= \mathcal{I}_{\text{ds}}(\alpha_1) \circ \mathcal{I}_{\text{ds}}(\alpha_2) \text{ by definition of } \mathcal{I}_{\text{ds}}. \end{aligned}$$

The result $\mathcal{I}_{\text{sos}}(\alpha) = \mathcal{I}_{\text{ds}}(\alpha)$ follows from the induction hypothesis.

Case 3. $\alpha = \alpha_1 \cup \alpha_2$.We have

$$\begin{aligned} \mathcal{I}_{\text{sos}}(\alpha_1 \cup \alpha_2) &= \mathcal{I}_{\text{sos}}(\alpha_1) \cup \mathcal{I}_{\text{sos}}(\alpha_2) \text{ by Lemma 3.13} \\ \mathcal{I}_{\text{ds}}(\alpha_1 \cup \alpha_2) &= \mathcal{I}_{\text{ds}}(\alpha_1) \cup \mathcal{I}_{\text{ds}}(\alpha_2) \text{ by definition of } \mathcal{I}_{\text{ds}}. \end{aligned}$$

Again, the result $\mathcal{I}_{\text{sos}}(\alpha) = \mathcal{I}_{\text{ds}}(\alpha)$ follows from the induction hypothesis.

Case 4. $\alpha = \alpha_1^*$. We have

$$\begin{aligned} \mathcal{I}_{\text{sos}}(\alpha_1^*) &= (\mathcal{I}_{\text{sos}}(\alpha_1))^* \text{ by Lemma 3.14} \\ \mathcal{I}_{\text{ds}}(\alpha_1^*) &= (\mathcal{I}_{\text{ds}}(\alpha_1))^* \text{ by definition of } \mathcal{I}_{\text{ds}}. \end{aligned}$$

And finally, the result $\mathcal{I}_{\text{sos}}(\alpha) = \mathcal{I}_{\text{ds}}(\alpha)$ follows from the induction hypothesis.

□

Chapter 4

Abstraction refinement

We will define an algorithm that can verify programs in propositional dynamic logic. To avoid the need to exhaustively check all states, this algorithm uses finite abstractions of the program. These abstractions are overapproximations of the program, which means that if the program is incorrect this information is present in the abstraction, see [5]. The abstraction can contain information that indicates that the program is incorrect while the program is actually correct. This is called a false counterexample. To distinguish between true and false counterexamples, the algorithm computes a formula that is satisfiable if and only if the counterexample is true and it uses a theorem prover to obtain the answer. When a counterexample is found to be false, this information is used to refine the abstraction. This method is called counterexample-driven refinement, see [4].

The algorithm terminates when a counterexample is found to be true or when there are no counterexamples present in the abstraction. In the latter case we can be sure that the program is correct, since the abstraction is an overapproximation.

Like the algorithm in [1] we start with an abstraction that abstracts away from everything and we only include information that is used to exclude a false counterexample.

In Section 4.1 we define the algorithm and refer to other sections for the definitions used in the algorithm. We conclude the chapter with Section 4.6 where we present the proof that the algorithm is correct.

4.1 Overview

Fix a signature $(\mathcal{P}, \mathcal{B})$ and a model M. Let α and $(\varphi_{pre}, \varphi_{post})$ be a program and its specification as defined in Definition 2.3. We require two properties of the model, these are explained in Section 4.2. Verifying the program means deciding whether $M \vDash \varphi_{pre} \rightarrow [\alpha]\varphi_{post}$ or not.

The algorithm is defined in Listing 4.1. Below the algorithm a short explanation and references to the formal definitions and proofs are given.

Listing 4.1 The abstraction refinement algorithm

The algorithm

1. Initialize the abstraction A as the control flow graph of α where each node is associated with the formula $\top$.
2. Split $(I, \top)$ into (I, φ_{pre}) and $(I, \neg\varphi_{pre})$. Split $(F, \top)$ into (F, φ_{post}) and $(F, \neg\varphi_{post})$.
3. Repeat the following
   (a) Find an abstract path $S_0, \ldots, S_n$ in A with $S_0 = (I, \varphi_{pre})$ and $S_n = (F, \neg\varphi_{post})$.
   (b) If such a path does not exist, output that α is correct. Else, continue.
   (c) For $i \in \{0, \ldots, n\}$ let ρ_i be the path constraint ρ_i of $S_0, \ldots, S_i$. Find the smallest i such that ρ_i is not satisfiable.
   (d) If such an i does not exist, output that α is incorrect. Continue otherwise.
   (e) Change the abstraction A by splitting along the path $S_0, \ldots, S_i$.

Explanation

1. An abstraction contains a finite partition of the state space for each node in the control flow graph. Partition classes are represented by formulas, see Definition 4.7 on page 24 for a formal definition. Initially all partitions contain one class represented by $\top$, this is the most abstract way of considering the program. See Definition 4.8 for the initial abstraction.
2. In Definition 4.14 on page 26 it is defined how to split nodes. The refinement in this step enables the algorithm to search for a path from (I, φ_{pre}) to $(F, \neg\varphi_{post})$ which could lead to a counterexample.
3. In each iteration the abstraction A is changed, or the algorithm terminates.
   (a) The abstraction A is a finite, directed graph. The nodes (I, φ_{pre}) and $(F, \neg\varphi_{post})$ will always exist, see Lemma 4.22 on page 29.
   (b) If the algorithm outputs that α is correct, the program α is indeed correct, see Theorem 4.26 on page 31.
   (c) Path constraints are defined in Definition 4.12 on page 26. Having a way to check satisfiability is an assumption given in Section 4.2.
   (d) If the algorithm outputs that α is incorrect, the program α is indeed incorrect, see Theorem 4.26.
   (e) Splitting along a path is defined in Definition 4.18 on page 29.

4.2 Assumptions

We require two things about the model M. We assume there is a way to determine the satisfiability of a formula $\varphi \in \Phi$ where φ does not contain program modalities. Note that since φ does not contain modalities, the satisfiability only depends on $\mathcal{V}$ and not on $\mathcal{R}$. Hence, the verification question $M \vDash \varphi_{pre} \to [\alpha]\varphi_{post}$ cannot be directly answered using this assumption.

Secondly, we assume that for every $b \in \mathcal{B}$ there exists a weakest precondition operator, that we will define below. We will use weakest preconditions to propagate constraints through the abstraction.

Definition 4.1 (Weakest preconditions)**.** Let $b \in \mathcal{B}$, a weakest precondition WP_b is a function from Φ to Φ with the following property. We have $s \vDash \mathrm{WP}_b(\varphi)$ if and only if there exists a $r \in \Sigma$ with $r \vDash \varphi$ and $(s, r) \in \mathcal{R}(b)$.

Example 4.2. We will prove that the model NDB we defined in section 2.3 satisfies these requirements. Recall that $\mathcal{P}^{\mathrm{NDB}}$ is a set of variables, so the set of formulas without modalities are the propositional logic formulas. Although the satisfiability problem of propositional logic is NP-hard, there is a way to determine satisfiability.

Let $\varphi[p \mapsto \psi]$ stand for the formula φ where all occurrences of p are replaced by the formula ψ. Then define the weakest precondition by

$$\mathrm{WP}_b(\varphi) = \begin{cases} \varphi[p \mapsto x] & \text{if } b = (p := x) \text{ with } x \in \mathit{bool} = \{\top, \bot\} \\ \varphi[p \to \top] \vee \varphi[p \to \bot] & \text{if } b = (p := \mathit{random})\,. \end{cases}$$

Proposition 4.3. *The function* WP_b *defined above with* b *of the form* $p := x$ *where* $x \in \{\top, \bot\}$ *is a weakest precondition.*

Proof. Let $s \in \Sigma$, by definition of $\mathcal{R}^{\mathrm{NDB}}(b)$ there is exactly one $r \in \Sigma$ with $(s, r) \in \mathcal{R}^{\mathrm{NDB}}(b)$. It is left to prove that $s \vDash \mathrm{WP}_b(\varphi)$ iff $r \vDash \varphi$. We prove this with induction to the structure of φ.

Case 1. $\varphi = \top$. We have $\mathrm{WP}_b(\top) = \top$, both s and r satisfy $\top$.

Case 2. $\varphi = p$. We have $\mathrm{WP}_b(p) = x$. By definition of r, we have $r(p) = x$, so $r \vDash p$ iff $x = \top$. Hence $s \vDash \mathrm{WP}_b(p)$ iff $r \vDash p$.

Case 3. $\varphi = q$, with $q \neq p$. We have $\mathrm{WP}_b(q) = q$. The result follows from $r(q) = s(q)$.

Case 4. $\varphi = \neg\psi$. We have $s \vDash \mathrm{WP}_b(\neg\psi) = \neg\mathrm{WP}_b(\psi)$ iff $s \nvDash \mathrm{WP}_b(\psi)$. By the induction hypothesis, this holds iff $r \nvDash \psi$. This is equivalent with $r \vDash \neg\psi$.

Case 5. $\varphi = \varphi_1 \wedge \varphi_2$. We have $\mathrm{WP}_b(\varphi_1 \wedge \varphi_2) = \mathrm{WP}_b(\varphi_1) \wedge \mathrm{WP}_b(\varphi_2)$. So $s \vDash \mathrm{WP}_b(\varphi)$ iff both $s \vDash \mathrm{WP}_b(\varphi_1)$ and $s \vDash \mathrm{WP}_b(\varphi_2)$. By the induction hypothesis, we have $s \vDash \mathrm{WP}_b(\varphi_i)$ iff $r \vDash \varphi_i$, for $i \in \{1, 2\}$. Then the result follows from $r \vDash \varphi$ iff both $r \vDash \varphi_1$ and $r \vDash \varphi_2$.

□

Proposition 4.4. *The function* WP_b *defined above with* b *of the form* $p := \mathit{random}$ *is a weakest precondition.*

Proof. We will reduce this problem to the previous Proposition. Let $s \in \Sigma$, by definition of $\mathcal{R}^{\mathrm{NDB}}(b)$ there are two states with $(s,r) \in \mathcal{R}^{\mathrm{NDB}}(b)$, one of them sends p to $\top$, the other to $\bot$. Let $r_1$be the former and r_0 the latter.

Suppose $s \vDash \mathrm{WP}_b(\varphi)$. Then $s \vDash \varphi[p \to \top]$ or $s \vDash \varphi[p \to \bot]$. Assume the former, consider the action $c \in \mathcal{B}$ with $c = (p := \top)$ we have $s \vDash \mathrm{WP}_c(\varphi)$. Hence there is a $r \in \Sigma$ with $(s,r) \in \mathcal{R}^{\mathrm{NDB}}(c)$ and $r \vDash \varphi$. From the definition of $\mathcal{R}^{\mathrm{NDB}}(c)$ we also have $(s,r) \in \mathcal{R}^{\mathrm{NDB}}(b)$. In the case $s \vDash \varphi[p \to \bot]$ the argument is similar, this concludes the proof of this direction.

Suppose there is a r with $r \vDash \varphi$ and $(s,r) \in \mathcal{R}^{\mathrm{NDB}}(b)$. Then $r = r_0$ or $r = r_1$. Assume the former, define the action $c \in \mathcal{B}$ with $c = (p := \bot)$. Since $(s, r_0) \in \mathcal{R}^{\mathrm{NDB}}(c)$ we have $s \vDash \mathrm{WP}_c(\varphi) = \varphi[p \mapsto \bot]$. Then also $s \vDash \mathrm{WP}_b(\varphi)$. When $r = r_1$ the argument is similar. □

A note on actions

It will be convenient to treat basic and test actions the same. We will do this in the same way as defined in Section 3.2.

Define the set of actions $\mathcal{A} = \mathcal{B} \cup \{?\varphi \mid \varphi \in \Phi\}$. Extend the interpretation $\mathcal{R}$ of basic actions $b \in \mathcal{B}$ to all actions by defining

$$\mathcal{R}(?\varphi) = \{(s,s) \mid s \vDash \varphi\}.$$

Using this extension of $\mathcal{R}$ we can generalize Definition 4.1 of weakest preconditions to all $a \in \mathcal{A}$.

Definition 4.5 (Weakest preconditions)**.** Let $a \in \mathcal{A}$, a function $\mathrm{WP}_a : \Phi \to \Phi$ is a weakest precondition if the following holds. We have $s \vDash \mathrm{WP}_a(\varphi)$ if and only if there exists a $r \in \Sigma$ with $r \vDash \varphi$ and $(s,r) \in \mathcal{R}(a)$.

We assumed that there is a weakest precondition operator WP_b for all $b \in \mathcal{B}$. We define $\mathrm{WP}_{?\psi}(\varphi) = \varphi \wedge \psi$ to have a weakest precondition of all actions $a \in \mathcal{A}$.

Proposition 4.6. *The operator* $\mathrm{WP}_{?\psi}$ *defined above is a weakest precondition.*

Proof. Let $s \in \Sigma$. Suppose $s \vDash \mathrm{WP}_{?\psi}(\varphi) = \varphi \wedge \psi$. Since $s \vDash \psi$ we have $(s,s) \in \mathcal{R}(?\psi)$ and we have $s \vDash \varphi$.

Suppose there is a $r \in \Sigma$ with $(s,r) \in \mathcal{R}(?\psi)$ and $r \vDash \varphi$. Then $s = r$ and $s \vDash \psi$. Hence, $s \vDash \mathrm{WP}_{?\psi}(\varphi)$. □

4.3 Abstractions

The abstraction we make is that we use a partition of the state space instead of the state space itself. A partition class S will be represented by a formula φ^S, we can see this a partition class by $s \in S$ iff $s \vDash \varphi^S$.

The control flow graph of the program will be used to represent the program α. Recall that nodes represent the stages during the computation of α. At different stages we need different information about the states, so for each node in the graph we maintain a separate partition of the state space.

The outgoing edges of a node represent the actions that are possible to take next, however not all these actions will result in a successful computation when executed at a state. We will use the partitions to denote the set of states where an action can be successfully executed on. To be able to represent this we want labeled edges between partition classes instead of between nodes in the control flow graph. This leads to the following definition. Note that this definition does not enforce that the partition classes for a certain node form a partition, but in our use of the abstraction this will be the case.

Definition 4.7 (Abstract programs)**.** Let α be a program, let G be its control flow graph and N^G the nodes of G. An abstraction A of α is a tuple (N, E) where $N \subseteq N^G \times \Phi$ is a set of nodes labeled by nodes from the control flow graph and formulas; and $E \subseteq N \times \mathcal{A} \times N$ a set of directed edges labeled with actions. We require that there are no edges of the form (S, a, T) with $S = T$.

In the initial abstraction we want to abstract away from everything. Therefore, it is the same as the control flow graph where each node is associated with the partition containing one class, represented by $\top$.

Definition 4.8 (Initial abstraction)**.** Let α be a program, let G be its control flow graph with nodes N^G and edges E^G. The initial abstraction $A = (N, E)$ of α is defined by

$$\begin{aligned} N &= \left\{(g, \top) \mid g \in N^G\right\} \\ E &= \left\{((g, \top), a, (h, \top)) \mid (g, a, h) \in E^G\right\}. \end{aligned}$$

Note that by Fact 3.9 on page 15 we have that there are no edges (S, a, T) with $S = T$, so this definition satisfies the requirement given in the definition of abstract programs.

Example 4.9. Recall the example program given in Listing 2.2 on page 10 and its control flow graph given in Figure 3.3 on page 16. The initial abstraction of this program is given in Figure 4.1 on the following page. Note that in all nodes other than $(I, \top)$ and $(F, \top)$ we abbreviated $(g, \top)$ to $\top$ since the node g is only used to distinguish the nodes.

We want to exclude the possibility of a false-positive. We do this by using abstractions that overapproximate.

Definition 4.10 (Overapproximation)**.** An abstraction $A = (N, E)$ of α is an overapproximation if the following holds. For each path γ in the control flow graph of α that has a satisfying trace σ, where

$$\begin{aligned} \gamma &= I \xrightarrow{a_1} g_1 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} g_{n-1} \xrightarrow{a_n} F \\ \sigma &= (s_0, \dots, s_n), \end{aligned}$$

there exist formulas $\psi_0, \dots, \psi_n$ such that $s_i \vDash \psi_i$, $(g_i, \psi_i) \in N$ and

$$((g_{i-1}, \psi_{i-1}), a_i, (g_i, \psi_i)) \in E.$$

Lemma 4.11. *The initial abstract program A is an overapproximation.*

Proof. Let γ be a path with nodes $g_0, \dots, g_n$ and edge-labels $a_1, \dots, a_n$ and let $(s_0, \dots, s_n)$ be a satisfying trace. Choose $\psi_i = \top$. We have $s_i \vDash \top$ and by definition $(g_i, \top) \in N$. Since γ is a path, we have by definition of E that $((g_{i-1}, \top), a_i, (g_i, \top)) \in E$. □

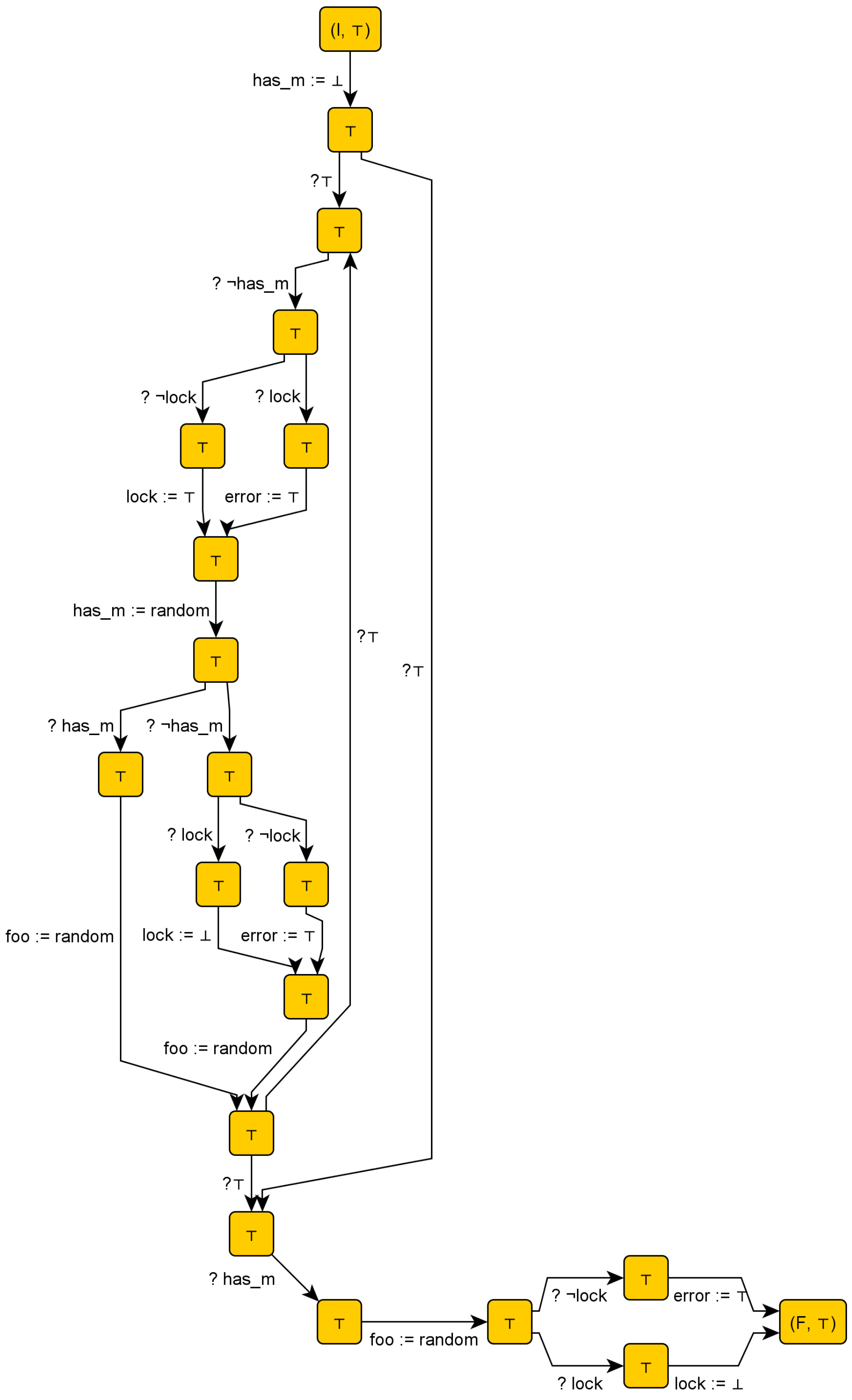


Figure 4.1: The initial abstraction of the lock/unlock program

4.4 Path constraints

By design the abstractions will contain many false counterexamples. A counterexample in the abstraction is false when there does not exist a concrete trace that follows the example.

To distinguish between true and false counterexamples, we will define path constraints. A path constraint of a path will be satisfiable if and only if the path is satisfiable by a concrete trace.

Definition 4.12 (Path constraints)**.** Let A be an abstract program and Γ a path through A with nodes $S_0, \ldots, S_n$ and edge-labels $a_1, \ldots, a_n$. The path constraint $\rho \in \Phi$ of Γ is defined with induction to n.

Suppose $n = 0$. Let $S_0 = (g, \varphi)$, we define $\rho = \varphi$.

For $n > 0$, let τ be the path constraint of the path with nodes $S_1, \ldots, S_n$ and edge-labels $a_2, \ldots, a_n$. Let $S_0 = (g, \varphi)$, then $\rho = \varphi \wedge \mathrm{WP}_{a_1}(\tau)$.

Lemma 4.13. *Let Γ be a path in A and γ in the control flow graph with*

$$\begin{aligned}\Gamma &= (g_0, \varphi_0) \xrightarrow{a_1} \ldots \xrightarrow{a_n} (g_n, \varphi_n) \\ \gamma &= g_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} g_n\end{aligned}$$

Let $s \in \Sigma$ and ρ the path constraint of Γ. We have $s \vDash \rho$ iff there exists trace $s_0, \ldots, s_n$ that satisfies γ with $s_0 = s$ and $s_i \vDash \varphi_i$.

Proof. We prove this with induction to n. Suppose $n = 0$. Since γ is a path without edges, all traces (s) satisfy γ. Because $\rho = \varphi_0$, we have $s \vDash \rho$ iff $s \vDash \varphi_0$.

Let $n > 0$. Let τ be the path constraint of the following path Γ', we have $\rho = \varphi_0 \wedge \mathrm{WP}_{a_1}(\tau)$.

$$\Gamma' = (g_1, \varphi_1) \xrightarrow{a_2} \ldots \xrightarrow{a_n} (g_n, \varphi_n).$$

Suppose that $s \vDash \rho$. We already get $s \vDash \varphi_0$. From the definition of weakest preconditions we have that there exists a r with $(s, r) \in \mathcal{R}(a_1)$ and $r \vDash \tau$. Using the induction hypothesis on $r \vDash \tau$ we obtain a trace $(s_1, \ldots, s_n)$ that satisfies γ' with $s_1 = r$ and $s_i \vDash \varphi_i$ for $i > 0$. Hence, $(s, s_1, \ldots, s_n)$ satisfies γ.

Suppose that there exists a trace $(s_0, \ldots, s_n)$ that satisfies γ and $s_i \vDash \varphi_i$. Using the induction hypothesis we have that $s_1 \vDash \tau$. Since $(s_0, s_1) \in \mathcal{R}(a_1)$ we have by the definition of weakest precondition that $s_0 \vDash \mathrm{WP}_{a_1}(\tau)$. From $s_0 \vDash \varphi_0$ it follows that $s_0 \vDash \rho$. □

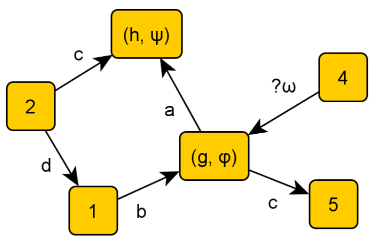
4.5 Refinements

Abstractions can be refined to include new information. When we want to distinguish between states that do or do not satisfy a property ψ, we can refine the partitions in an abstraction in the following way. See Figure 4.2 on the next page for an example.

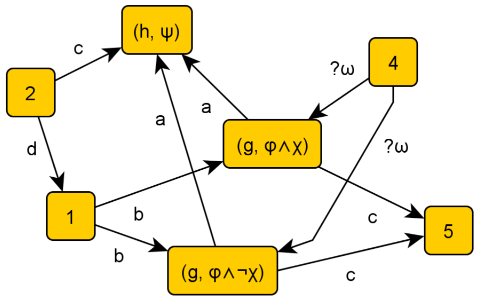
Definition 4.14 (Splitting a node)**.** Let $A = (N, E)$ be an abstract program, $S = (g, \varphi)$ a node in A and ψ a formula that will be used to split S. We define the abstraction $A' = (N', E')$ that is the result of splitting S by ψ as follows.

Define $S^- = (g, \varphi \wedge \neg\psi)$ and $S^+ = (g, \varphi \wedge \psi)$. The nodes of A' are defined by

$$N' = N \cup \{S^-, S^+\} \setminus \{S\}.$$



(a) Before splitting



(b) After splitting

Figure 4.2: Splitting the node (g, φ) using χ

We do not want to change anything essential about the edges. If S is part of an edge in E, then we want two edges in E', one where S^- has the same role as S and one with S^+ in the same role. By definition an abstraction does not contain an edge from a node to itself, so the edge (S, a, S) does not exist in E. Using this, we can define the edges E' by

$$\begin{aligned} E' &= E \cap (N' \times \mathcal{A} \times N') \\ &\cup \big\{ \big(S^-, a, T\big) \mid (S, a, T) \in E \big\} \\ &\cup \big\{ \big(S^+, a, T\big) \mid (S, a, T) \in E \big\} \\ &\cup \big\{ \big(T, a, S^-\big) \mid (T, a, S) \in E \big\} \\ &\cup \big\{ \big(T, a, S^+\big) \mid (T, a, S) \in E \big\} . \end{aligned}$$

Note that splitting a node does not introduce edges of the form (T, a, T). Hence, A' is an abstraction.

Lemma 4.15. *Let A and A' be abstractions where A' is obtained from A by splitting a node. If A is an overapproximation, then A' is an overapproximation.*

Proof. Let $S = (g, \varphi)$ be the node that has been split using a formula ψ. Let γ be a path with satisfying trace σ

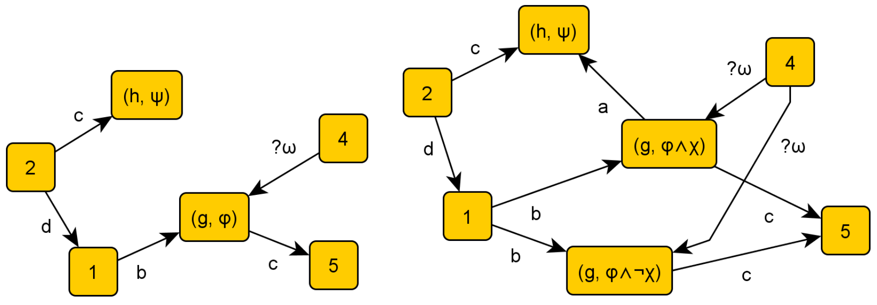
$$\begin{aligned} \gamma &= h_0 \overset{a_1}{\to} \dots \overset{a_n}{\to} h_n \\ \sigma &= (s_0, \dots, s_n). \end{aligned}$$

Since A is an overapproximation, there exists formula χ_i such that $s_i \vDash \chi_i$ and the path Γ given below is a path in the abstraction.

$$\Gamma = S_0 \overset{a_1}{\to} \dots \overset{a_n}{\to} S_n \text{ with } S_i = (h_i, \chi_i).$$

We will replace each node S in Γ to obtain a path Γ' in A'. The edge-labels of Γ' will be the same as in Γ, we construct the nodes S_i' as follows. If $S_i \neq S$ then $S_i' = S_i$. Else if $S_i = S$ and $s_i \vDash \psi$ then we take $S_i' = S^+ = (g, \varphi \wedge \psi)$. Otherwise, $S_i' = S^- = (g, \varphi \wedge \neg\psi)$.

By the definition of the edges of A', we have that Γ' is a path in A'. With χ_i' the formulas of the nodes S_i', we have $s_i \vDash \chi_i'$ since we chose S^+ when $s_i \vDash \psi$ and S^- otherwise. We conclude that A' is an overapproximation. □



(a) When $\chi \wedge \varphi$ is not satisfiable (b) When $\chi \wedge \varphi$ is satisfiable

Figure 4.3: Splitting the transition from (g, φ) to (h, ψ) with edge-label a

When the algorithm has encountered a false counterexample, we want to incorporate this information in the abstraction. A false counterexample is a path in the abstraction from (I, φ_{pre}) to $(F, \neg\varphi_{post})$ that cannot be satisfied by a concrete trace. Hence, we want to delete this path in the abstraction. Since the transitions of this path can be used by other paths that we do not want to delete, we will use the weakest precondition of an action to split the nodes along this path and delete the edges for which we are sure that no path can take them.

First we define how to split along one transition, then we will define how to split along a full path using the first definition. For both we will prove that an overapproximation will stay an overapproximation.

Definition 4.16 (Splitting along a transition)**.** Let $S = (g, \varphi)$ and $T = (h, \psi)$ be two nodes in the abstract program A with an edge from S to T labeled with $a \in \mathcal{A}$. See Figure 4.2a on the preceding page for an illustration. We obtain a new abstract program by splitting this transition as follows.

Let χ be $\mathrm{WP}_a(\psi)$. We check whether $\chi \wedge \varphi$ is satisfiable. If it is not, let the resulting abstraction be the same as A except that the edge from S to T with label a is removed. The resulting abstraction is given in Figure 4.3a.

Else, let A' be the abstract program where (g, φ) is split using χ according to the previous definition. Let $S^- = (g, \varphi \wedge \neg\chi)$ and $S^+ = (g, \varphi \wedge \chi)$ be the new nodes. Then let the resulting abstraction be A' where the edge from S^- to T is removed. See Figure 4.3b for an illustration of the resulting abstraction.

Lemma 4.17. *Let A and A' be abstractions where A' is obtained from A by splitting along a transition. If A is an overapproximation, then A' is an overapproximation.*

Proof. Let the transition from (g, φ) to (h, ψ) with edge-label a be the transition that has been split. Define $\chi = \mathrm{WP}_a(\psi)$.

Suppose $\chi \wedge \varphi$ is satisfiable. Then the node (g, φ) is first split using χ. This results in an abstraction B that is an overapproximation by Lemma 4.15.

Let γ be a path in the control flow graph that is satisfied by a trace σ

$$\begin{aligned}\gamma &= g_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} g_n \\ \sigma &= (s_0, \dots, s_n).\end{aligned}$$

Since B is an overapproximation, there are formulas ψ_i such that $s_i \vDash \psi_i$ and Γ defined below is path in the abstraction B.

$$\Gamma = S_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} S_n \text{ with } S_i = (g_i, \psi_i).$$

To see that Γ is also a path in A', assume otherwise. Hence, there exists an i with $S_i = (g, \varphi \wedge \neg\chi)$, $S_{i+1} = (h, \psi)$ and $a_{i+1} = a$. Since $(s_i, s_{i+1}) \in \mathcal{R}(a)$ and $s_{i+1} \vDash \psi$, we have $s_i \vDash \mathrm{WP}_a(\psi) = \chi$ by the definition of weakest precondition. This contradicts with $s_i \vDash \neg\chi$. We conclude that A' is an overapproximation.

Otherwise, $\chi \wedge \varphi$ is not satisfiable. The proof is similar to the second part of the previous case, namely take $B = A$. □

Definition 4.18 (Splitting along a path)**.** Let $S_0, \dots, S_n$ be a path in the abstract program A with edge-labels $a_1, \dots, a_n$. First we split A along the transition S_{n-1}, S_n with label a_n. If this split resulted in new states S^+ and S^-, and we have $n > 1$, then we recursively split along the path $S_0, \dots, S_{n-1}, S^+$ with edge labels $a_1, \dots, a_{n-1}$. Else we are finished.

Lemma 4.19. *Let A and A' be abstractions where A' is obtained from A by splitting along a path. If A is an overapproximation, then A' is an overapproximation.*

Proof. Splitting along a path is defined by repeatedly splitting along a transition. The result follows from repeatedly applying Lemma 4.17. □

The abstraction refinement algorithm will search for a path from (I, φ_{pre}) to $(F, \neg\varphi_{post})$ in each iteration. The following arguments show that these nodes will always be present in the abstraction.

Fact 4.20. *By definition the last node S_n is not split when splitting along the path $S_0, \dots, S_n$.*

Lemma 4.21. *Let Γ be a path with nodes $S_0, \dots, S_n$ whose path constraint is unsatisfiable. Splitting along this path does not split the first node S_0.*

Proof. Let $S_i = (g_i, \varphi_i)$ and let $a_1, \dots, a_n$ be the edge-labels of the path Γ. We prove the contraposition of the Lemma.

Assume S_0 has been split. Then all nodes S_i with $i < n$ are also split, let S_i^- and $S_i^+ = (g_i, \varphi_i^+)$ be the resulting states. Define $\varphi_n^+ = \varphi_n$. We will prove with (downward) induction to i that φ_i^+ is the path constraint ρ_i of the path $S_i, \dots, S_n$.

Let $i = n$. The path constraint ρ_n of the path S_n is φ_n which is by definition φ_n^+.

Let $i < n$. The formula used to split S_i is $\mathrm{WP}_{a_{i+1}}(\varphi_{i+1}^+)$, so $\varphi_i^+ = \varphi_i \wedge \mathrm{WP}_{a_i}(\varphi_{i+1}^+)$. The path constraint ρ_i is defined as $\rho_i = \varphi_i \wedge \mathrm{WP}_{a_{i+1}}(\rho_{i+1})$. By the induction hypothesis, $\rho_{i+1} = \varphi_{i+1}^+$. Hence, $\varphi_i^+ = \rho_i$. This concludes the induction proof.

Since splitting the transition from S_0 to S_1^+ resulted in S_0 being split, we have by Definition 4.16 that $\varphi_0 \wedge \mathrm{WP}_{a_1}(\varphi_1^+) = \varphi_0^+$ is satisfiable. Hence, the path constraint ρ_0 of the path $S_0, \dots, S_n$ is satisfiable. □

Lemma 4.22. *At step 3(a) of the abstraction refinement algorithm (see Listing 4.1 on page 21) there exist nodes (I, φ_{pre}) and $(F, \neg\varphi_{post})$ in the abstraction.*

Proof. The nodes (I, φ_{pre}) and $(F, \neg\varphi_{post})$ are introduced in the abstraction at step 2. The only place where the abstraction changes after step 2 is in step 3(e).

The node $(F, \neg\varphi_{post})$ has no outgoing edges, so it can only occur as the last node in a path. By Fact 4.20 we have that $(F, \neg\varphi_{post})$ will always be present in the abstraction.

The node (I, φ_{pre}) has no incoming edges, so it can only occur as the first node in a path. In the algorithm paths are only split when their path constraint is unsatisfiable, the result follows from Lemma 4.21. $\square$

4.6 Correctness

We will first prove that when α does not satisfy its specification, there will always exist a (true or false) counterexample in overapproximations of α. Then we prove that if a counterexample has a satisfiable path constraint, α does not satisfy its specification. Observing that the abstractions used in the abstraction refinement algorithm are always overapproximations, we are then able to prove the correctness of the algorithm itself.

Lemma 4.23. *If $M \nvDash \varphi_{pre} \to [\alpha]\varphi_{post}$ and A is an overapproximation of α, then there exists a path Γ in A that starts at (I, φ_{pre}) and ends at $(F, \neg\varphi_{post})$.*

Proof. There exists a $s \in \Sigma$ with $s \vDash \varphi_{pre} \wedge \langle\alpha\rangle\neg\varphi_{post}$. By the structural operational semantics there is a trace $(r_0, \ldots, r_n)$ with $s = r_0$ and $r_n \vDash \neg\varphi_{post}$ that satisfies a path γ from I to F in the control flow graph of α . Let $g_0, \ldots, g_n$ be the nodes of γ.

Since A is an overapproximation there exists a path Γ with nodes (g_i, ψ_i) such that $r_i \vDash \psi_i$. There are only two nodes in A of the form (I, ψ), namely (I, φ_{pre}) and $(I, \neg\varphi_{pre})$. Since $g_0 = I$, $r_0 \vDash \varphi_{pre}$ we have $\psi_0 = \varphi_{pre}$.

Similarly, there are only two nodes of the form (F, ψ), namely (F, φ_{post}) and $(F, \neg\varphi_{post})$. Since $g_n = F$, $r_n \vDash \neg\varphi_{post}$ we have $\psi_n = \neg\varphi_{post}$. Hence, Γ starts at (I, φ_{pre}) and ends at $(F, \neg\varphi_{post})$. $\square$

Lemma 4.24. *If the path constraint ρ of a path Γ that starts at (I, φ_{pre}) and ends at $(F, \neg\varphi_{post})$ is satisfiable, then $M \nvDash \varphi_{pre} \to [\alpha]\varphi_{post}$.*

Proof. Let $(g_0, \varphi_0), \ldots, (g_n, \varphi_n)$ be the nodes of Γ and $a_1, \ldots, a_n$ its edge-labels. Since the path constraint is satisfiable, there exists a $s \in \Sigma$ with $s \vDash \rho$. By Lemma 4.13 there is a trace $s_0, \ldots, s_n$ that satisfies the path with nodes $g_0, \ldots, g_n$ and edge-labels $a_1, \ldots, a_n$ and $s_0 \vDash \varphi_{pre}$ and $s_n \vDash \neg\varphi_{post}$. By the definition of structural operational semantics we have $(s_0, s_n) \in \mathcal{I}_{\text{sos}}(\alpha)$. Hence, $s_0 \vDash \varphi_{pre} \wedge \langle\alpha\rangle\neg\varphi_{post}$ so $M \nvDash \varphi_{pre} \to [\alpha]\varphi_{post}$. $\square$

Lemma 4.25. *The abstract program A is an overapproximation at every stage of the algorithm.*

Proof. When A is initialized, it is an overapproximation by Lemma 4.11. The abstraction A is changed in step (2) in the algorithm, see Listing 4.1. It stays an overapproximation by Lemma 4.15.

The only other step where A is changed in step (3e) of the algorithm. From Lemma 4.19 we can conclude that A will also remain an overapproximation after this step. $\square$

Theorem 4.26. *If the algorithm terminates, its output is correct.*

Proof. The algorithm can terminate at two places. Suppose the algorithm terminates at step (3b) outputting that α is correct. This means that there is no path Γ in abstraction A of that stage that starts at (I, φ_{pre}) and ends at $(F, \neg\varphi_{post})$. By Lemma 4.25 we have that the abstraction A of that stage is an overapproximation. By the contraposition of Lemma 4.23 we have $M \vDash \varphi_{pre} \to [\alpha]\varphi_{post}$, so α is indeed correct.

Suppose the algorithm terminates at step (3e) outputting that α is incorrect. Then the path constraint of a path Γ that starts at (I, φ_{pre}) and ends at $(F, \neg\varphi_{post})$ is satisfiable. From Lemma 4.24 it follows that $M \nvDash \varphi_{pre} \to [\alpha]\varphi_{post}$ so α is indeed incorrect. □

Chapter 5

Deterministic propositional dynamic logic

Propositional dynamic logic allows for non-determinism in two ways. The code constructs $\alpha_1 \cup \alpha_2$ and α^* are non-deterministic and actions can result in different states. An example of the last category is the action $p := \textit{random}$ defined in the NDB language in Section 2.3.

Most imperative languages only have deterministic code constructs and most statements execute deterministically. This can be used to make the abstraction refinement algorithm more efficient and it will also enable us to give the following termination result. When a deterministic program does not satisfy its specification, the abstraction refinement algorithm will eventually terminate and output that the program is incorrect.

Although non-deterministic actions do not usually occur in programming languages, they are still useful to model the environment of a program. We shall therefore allow arbitrary mixes between deterministic and non-deterministic actions and we will adapt the abstraction refinement algorithm to exploit determinism when present. Note that the termination result only holds when all actions are deterministic.

In Section 5.1 we define the language and semantics of a fragment of propositional dynamic logic that only contains deterministic code constructs. Then in Section 5.2 we formally define the two ways in which a program can be deterministic and we prove that the fragment of PDL we have defined satisfies this definition.

Then in Section 5.3 we adapt the abstraction refinement algorithm to make use of determinism when present. This allows us to prove the termination result in Section 5.4.

Finally, we run this version of abstraction refinement on the program we presented in the introduction. The results of this verification are discussed in Section 5.5.

5.1 Language and semantics

The language consists again of formulas and programs. Instead of non-determinism and looping we have if-then-else statements and while statements.

Definition 5.1. Let $(\mathcal{P}, \mathcal{A})$ be a signature. Define the deterministic PDL formulas Φ_d and programs Π_d by simultaneous recursion. Let $p \in \mathcal{P}$, $b \in \mathcal{B}$, $\varphi_1, \varphi_2 \in \Phi_\mathrm{d}$ and $\delta_1, \delta_2 \in \Pi_\mathrm{d}$, then

$$\begin{aligned} \varphi &::= \top \mid p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \langle\delta_1\rangle\varphi_1 \\ \delta &::= b \mid \delta_1 ; \delta_2 \mid \mathit{if}\ (\varphi)\ \{\delta_1\}\ \mathit{else}\ \{\delta_2\} \mid \mathit{while}\ (\varphi)\ \{\delta_1\}. \end{aligned}$$

We use the same models for D-PDL as we use for PDL, see Definition 2.4 on page 6. We give the denotational semantics by defining a translation π from D-PDL to PDL. We define π by recursion, the only interesting cases are the new constructs, for completeness we also give the other cases.

$$\begin{aligned} \pi(\top) &= \top \\ \pi(p) &= p \\ \pi(\neg\varphi) &= \neg\pi(\varphi) \\ \pi(\varphi_1 \wedge \varphi_2) &= \pi(\varphi_1) \wedge \pi(\varphi_2) \\ \pi(\langle\delta\rangle\varphi) &= \langle\pi(\delta)\rangle\pi(\varphi). \end{aligned}$$

$$\begin{aligned} \pi(b) &= b \\ \pi(\delta_1;\delta_2) &= \pi(\delta_1);\pi(\delta_2) \\ \pi(\mathit{if}\ (\varphi)\ \{\delta_1\}\ \mathit{else}\ \{\delta_2\}) &= (?\varphi;\pi(\delta_1)) \cup (?\neg\varphi;\pi(\delta_2)) \\ \pi(\mathit{while}\ (\varphi)\ \{\delta_1\}) &= (?\varphi;\pi(\delta_1))^* ; ?\neg\varphi. \end{aligned}$$

Recall that $\mathcal{I}_\mathrm{ds}$ is the notation for the denotational semantics of PDL programs and $\mathcal{I}_\mathrm{sos}$ for the structural operational semantics. For D-DPL programs we use the notation $\mathcal{I}_\mathrm{d\,ds}$ and $\mathcal{I}_\mathrm{d\,sos}$ for the two semantics.

Definition 5.2 (Denotational semantics)**.** The denotational semantics of $\delta \in \Pi_\mathrm{d}$ is defined by $\mathcal{I}^M_\mathrm{d\,ds}(\delta) = \mathcal{I}^M_\mathrm{ds}(\pi(\delta))$.

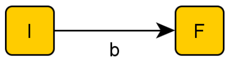
We could define the control flow graph of deterministic programs using the translation to PDL. However, in the case of the while statement it would not be clear from the control flow graph that this statement is deterministic. We therefore define the control flow graph of deterministic programs separately. Recall that $G(\alpha)$ is the notation for the control flow graph of a PDL program, we will use $G_\mathrm{d}(\delta)$ for the control flow graph of a D-PDL program.

Definition 5.3. Let $\delta \in \Pi_\mathrm{d}$ be a program. We define its control flow graph $G_\mathrm{d}(\delta)$ with induction to the structure of δ. See Figure 5.1 on the next page for an illustration.

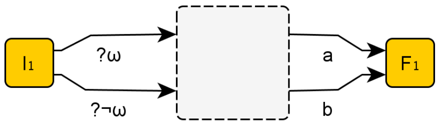
Case 1. $\delta = b$. Define $G_\mathrm{d}(b)$ as the graph with two nodes I and F, with an edge from I to F labeled with the action b.

Case 2. Recursive cases. Let $G_1 = G_\mathrm{d}(\delta_1)$ and $G_2 = G_\mathrm{d}(\delta_2)$ be the control flow graphs of δ_1 and δ_2 and let I_1, I_2 and F_1, F_2 denote their respective initial and final nodes.

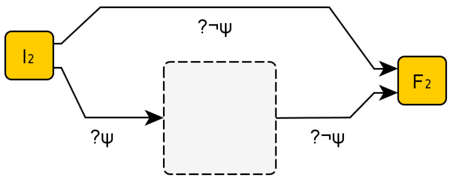
Case i. $\delta = \delta_1;\delta_2$. First define G as the disjoint union of G_1 and G_2. Set $I = I_1$ and $F = F_2$ and merge the node F_1 with I_2.



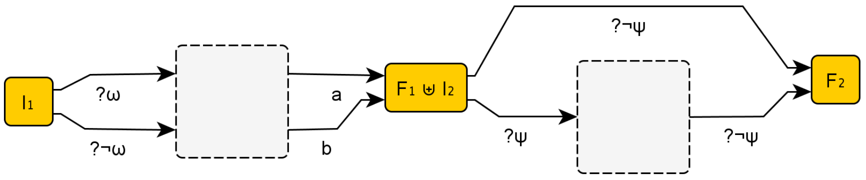
(a) The control flow graph of b



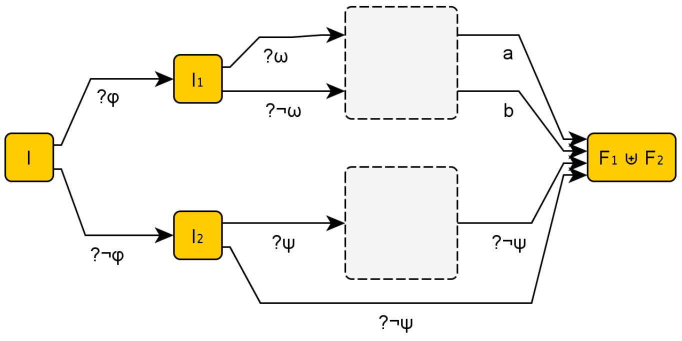
(b) The control flow graph of δ_1



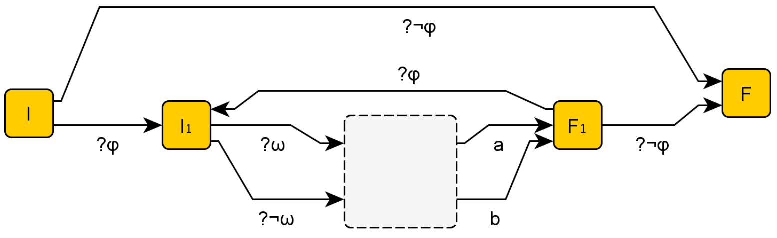
(c) The control flow graph of δ_2



(d) The control flow graph of $\delta_1; \delta_2$



(e) The control flow graph of *if* (φ) $\{\delta_1\}$ *else* $\{\delta_2\}$



(f) The control flow graph of *while* (φ) $\{\delta_1\}$

Figure 5.1: Control flow graphs of deterministic programs

Case ii. $\delta = \mathit{if}\ (\varphi)\ \{\delta_1\}\ \mathit{else}\ \{\delta_2\}$. Define G as the disjoint union of G_1, G_2 and a new node I. Set $F = F_1$ and merge the nodes F_1 with F_2. Add an edge from I to I_1 labeled with $?\varphi$ and add an edge from I to I_2 labeled with $?\neg\varphi$.

Case iii. $\delta = \mathit{while}\ (\varphi)\ \{\delta_1\}$. Define G as the disjoint union of G_1 with two new nodes I and F. Add the edges $I \to I_1$ and $F_1 \to I_1$ both with label $?\varphi$, and add the edges $I \to F$ and $F_1 \to F$ both with label $?\neg\varphi$.

Like PDL programs, we define the structural operational semantics using control flow graphs.

Definition 5.4 (Structural operational semantics)**.** Let M be a PDL-model. Define the GFC-model K in the same way as in Definition 3.10 on page 15. Then define

$$\mathcal{I}^M_{\mathrm{d\,sos}}(\delta) = \mathcal{I}^K_{\mathrm{graph}}(G_{\mathrm{d}}(\delta)).$$

Proposition 5.5. *Let M be a PDL-model. The structural operational interpretation $\mathcal{I}^M_{\mathrm{d\,sos}}(\delta)$ equals the denotational interpretation $\mathcal{I}^M_{\mathrm{d\,ds}}(\delta)$.*

Proof. Let K be the GFC-model as defined in Definition 3.10 on page 15. The following equalities hold

$$\begin{array}{rcl}
\mathcal{I}^M_{\mathrm{d\,ds}}(\delta) & = & \mathcal{I}^M_{\mathrm{ds}}(\pi(\delta)) \text{ by definition of } \mathcal{I}_{\mathrm{d\,ds}} \\
 & = & \mathcal{I}^M_{\mathrm{sos}}(\pi(\delta)) \text{ by Theorem 3.15} \\
 & = & \mathcal{I}^K_{\mathrm{graph}}\left(G(\pi(\delta))\right) \text{ by definition of } \mathcal{I}_{\mathrm{sos}} \\
\mathcal{I}^M_{\mathrm{d\,sos}}(\delta) & = & \mathcal{I}^K_{\mathrm{graph}}(G_{\mathrm{d}}(\delta)). \text{ by definition of } \mathcal{I}_{\mathrm{d\,sos}}.
\end{array}$$

We will prove with induction to the structure of δ that a trace $(s_0, \ldots, s_n)$ satisfies a path in $G(\pi(\delta))$ if and only if it satisfies a path in $G_{\mathrm{d}}(\delta)$. The while-statement is the only case where $G_{\mathrm{d}}(\delta)$ has a different structure than $G(\pi(\delta))$, this will be the most interesting case.

Case 1. $\delta = b$. We have $\pi(b) = b$ and $G(b) = G_{\mathrm{d}}(b)$.

Case 2. $\delta = \delta_1; \delta_2$. We have $\pi(\delta) = \pi(\delta_1); \pi(\delta_2)$. The graph $G(\pi(\delta))$ is in the same way constructed from $G(\pi(\delta_1))$ and $G(\pi(\delta_2))$ as $G_{\mathrm{d}}(\delta)$ is constructed from $G_{\mathrm{d}}(\delta_1)$ and $G_{\mathrm{d}}(\delta_2)$. The result follows from the induction hypothesis.

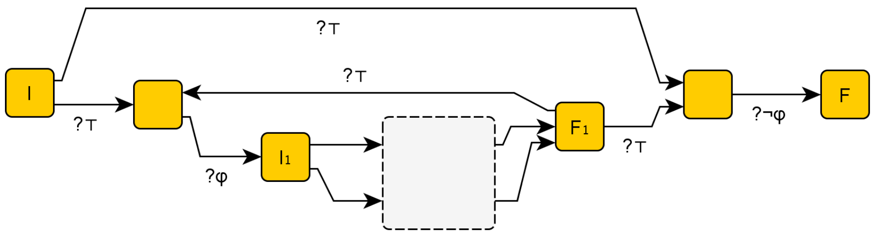
Case 3. $\delta = \mathit{if}\ (\varphi)\ \{\delta_1\}\ \mathit{else}\ \{\delta_2\}$. We have

$$G\left(\pi(\delta)\right) = G\left((?\varphi; \pi(\delta_1)) \cup (?\neg\varphi; \pi(\delta_2))\right).$$

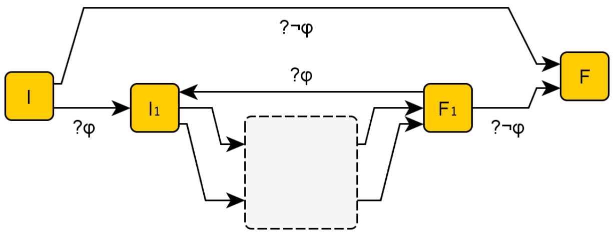
This graph is in the same way constructed from $G(\pi(\delta_1))$ and $G(\pi(\delta_2))$ as $G_{\mathrm{d}}(\delta)$ is constructed from $G_{\mathrm{d}}(\delta_1)$ and $G_{\mathrm{d}}(\delta_2)$. The result follows from the induction hypothesis.

Case 4. $\delta = \mathit{while}\ (\varphi)\ \{\delta_1\}$. We have

$$G\left(\pi(\delta)\right) = G\left((?\varphi; \pi(\delta_1))^*; ?\neg\varphi\right).$$



(a) The control flow graph $G\left(\pi\left(\textit{while}\;(\varphi)\;\{\delta_1\}\right)\right)$



(b) The control flow graph $G_{\mathrm{d}}\left(\textit{while}\;(\varphi)\;\{\delta_1\}\right)$

Figure 5.2: Equivalent control flow graphs

See Figure 5.2 for an illustration. From the figure it follows that every path in $G_{\mathrm{d}}(\delta)$ corresponds with a path in $G(\pi(\delta))$ by inserting edges $?\top$, and every path in $G(\pi(\delta))$ corresponds with a path in $G_{\mathrm{d}}(\delta)$ by removing edges $?\top$. Since $\mathcal{R}(?\top) = \{(s, s) \mid s \in \Sigma\}$, a satisfying trace of a path in $G_{\mathrm{d}}(\delta)$ corresponds with a satisfying trace of the corresponding path in $G(\pi(\delta))$ by the induction hypothesis and doubling states s in the trace when an edge $?\top$ is inserted. And in the other direction, a satisfying trace of a path in $G(\pi(\delta))$ corresponds with a satisfying trace of the corresponding path in $G_{\mathrm{d}}(\delta)$ by the induction hypothesis and merging two adjacent states s, s in the trace to s for every removed edge $?\top$.

□

5.2 Determinism

There can be two different types of determinism in a program. One is that at every stage there is only one statement that can be executed next, the other is that a single statement can only result in one state.

We define this in the languages of control flow graphs, let $\mathcal{A}$ be a CFG-signature and $K = (\Sigma, \mathcal{R})$ a CFG-model.

Definition 5.6 (Deterministic control flow graphs)**.** A control flow graph G is deterministic in K if the following holds. Let $s \in \Sigma$ be a state, g a node in G and $e_1, \ldots, e_n$ be the outgoing edges of g. Then there is at most one i such that there is a $r \in \Sigma$ with $(s, r) \in \mathcal{R}(a)$ where a is the label of e_i.

Definition 5.7 (Deterministic relation)**.** A relation $R \subseteq \Sigma \times \Sigma$ is deterministic if it is a partial function. That is, for every $s \in \Sigma$ there is at most one $r \in \Sigma$ with $(s, r) \in R$.

Definition 5.8 (Deterministic action)**.** An action $a \in \mathcal{A}$ is deterministic in K if the relation $\mathcal{R}(a)$ is deterministic.

We will prove that the control flow graphs of D-DPL programs are deterministic control flow graphs. First observe the following from the definition of G_{d}.

Fact 5.9. *Let G be the control flow graph of a program δ, then the node labeled F has no outgoing edges.*

Lemma 5.10. *The graph $G_{\mathrm{d}}(\delta)$ is a deterministic control flow graph.*

Proof. We prove this with structural induction on δ. Let $s \in \Sigma$ be a state, g a node in $G_{\mathrm{d}}(\delta)$ and $e_1, \dots, e_n$ be the outgoing edges of g.

Case 1. $\delta = b$. Then $G_{\mathrm{d}}(\delta)$ has only one edge.

Case 2. $\delta = \delta_1; \delta_2$. If $g \neq F_2 \uplus I_2$, then g and its outgoing edges are fully contained in either $G_{\mathrm{d}}(\delta_1)$ or $G_{\mathrm{d}}(\delta_2)$. If $g = F_2 \uplus I_2$, we have by Fact 5.9 that the outgoing edges of g are contained in $G_{\mathrm{d}}(\delta_2)$. In both cases the result follows from the induction hypothesis.

Case 3. $\delta = \mathit{if}\ (\varphi)\ \{\delta_1\}\ \mathit{else}\ \{\delta_2\}$. When $g = F$, it has no outgoing edges. When $g = I$ the outgoing edges are $?\varphi$ and $?\neg\varphi$. If $s \vDash \varphi$ there cannot be a r with $(s, r) \in \mathcal{R}(\neg\varphi)$ and when $s \nvDash \varphi$ there cannot be a r with $(s, r) \in \mathcal{R}(?\varphi)$. Hence, there is at most one i with the asked property. If $g \neq F$ and $g \neq I$, we have that g and its outgoing edges are fully contained in either $G_{\mathrm{d}}(\delta_1)$ or $G_{\mathrm{d}}(\delta_2)$. Then the result follows from the induction hypothesis.

Case 4. $\delta = \mathit{while}\ (\varphi)\ \{\delta_1\}$. When $g = F$, it has no outgoing edges. When $g = I$ or $g = F_1$ the outgoing edges are $?\varphi$ and $?\neg\varphi$. The argument is the same as in the previous case. Otherwise, we have that g and its outgoing edges are fully contained in $G_{\mathrm{d}}(\delta_1)$ and the result follows from the induction hypothesis.

□

To connect D-DPL with the second type of determinism, we prove that if all actions are deterministic the interpretation of a D-DPL program is deterministic.

Lemma 5.11. *Suppose all actions $b \in \mathcal{B}$ are deterministic in M. Let γ_1 and γ_2 be two paths in $G_{\mathrm{d}}(\delta)$ of length n that start at a node g. Let $s \in \Sigma$, let σ_1 and σ_2 be two traces starting with s that satisfy respectively γ_1 and γ_2. Then $\sigma_1 = \sigma_2$ and $\gamma_1 = \gamma_2$.*

Proof. We prove this with induction to n. If $n = 1$ then $\sigma_1 = \sigma_2$ since they both start with s.

Let $n > 1$. Let g_1 be the second node of γ_1, r_1 the second state of σ_1 and a_1 the label of the edge from g to g_1. Define g_2, a_2 and r_2 analogously. Since

$(s, r_1) \in \mathcal{R}(a_1)$ and $(s, r_2) \in \mathcal{R}(a_2)$ and G_d is a deterministic control flow graph by Lemma 5.10, we have that $g_1 = g_2$ and $a_1 = a_2$. If a_1 is a test action, we have $s = r_1 = r_2$. Else, a_1 is a basic action and by assumption deterministic. Hence, $r_1 = r_2$. The result follows from the induction hypothesis. □

Proposition 5.12. *Suppose all actions $b \in \mathcal{B}$ are deterministic in M. Let δ be a program, then its interpretation $\mathcal{I}^M(\delta)$ is a deterministic relation.*

Proof. Suppose $(s, r_1) \in \mathcal{I}^M(\delta)$, $(s, r_2) \in \mathcal{I}^M(\delta)$. By the structural operational semantics, there are paths γ_1, γ_2 in the graph $G_\mathrm{d}(\delta)$ that are satisfied by traces σ_1 and σ_2, where both traces start with s, σ_1 ends with r_1 and σ_2 ends with r_2.

Let n_1, n_2 be their lengths. Suppose $n_1 \leq n_2$. Look at the first n_1 nodes of γ_2. By Lemma 5.11 we have that this segment equals γ_1. Hence, the n_1-th node of γ_2 is F. By Fact 5.9 F has no outgoing edges and we conclude that $n_1 = n_2$. When $n_1 \geq n_2$ the argument is the same, hence $n_1 = n_2$. We apply Lemma 5.11 again to see that $r_1 = r_2$. □

5.3 Partition refinement on D-PDL

The partition refinement algorithm defined in Chapter 2 is defined over PDL programs $\alpha \in \Pi$. However, it only uses the control flow graph $G(\alpha)$ to verify α and not the specific PDL constructs. When we replace $G(\alpha)$ by $G_\mathrm{d}(\delta)$, we can therefore use the same definitions and proofs to define the algorithm on D-PDL.

The second change we make in the algorithm is that we use a different way of splitting along a path. This definition makes the algorithm more efficient when it splits along a path that contains deterministic actions.

The last change is that the algorithm searches for the shortest abstract path from (I, φ_{pre}) to $(F, \neg\varphi_{post})$ instead of an arbitrary path. This enables us to give a proof that the algorithm terminates on incorrect programs.

The algorithm is given in Listing 5.1 on the next page together with references to the relevant definitions and proofs.

Refinements

When we know that an action is deterministic, we can remove more edges when we split along a transition labeled by that action. See Figure 5.3 on page 40 for an illustration of all the possible cases.

Definition 5.13 (Splitting along a transition)**.** Let $S = (g, \varphi)$ and $T = (h, \psi)$ be two nodes in the abstract program A with an edge from S to T labeled with $a \in \mathcal{A}$. We obtain a new abstract program by splitting this transition as follows.

First we refine the abstraction according to Definition 4.16 on page 28. Recall that when $\chi \wedge \varphi$ is not satisfiable where $\chi = \mathrm{WP}_a(\psi)$, this results in an abstraction as shown in Figure 5.3c. When it is satisfiable it results in an abstraction as shown in Figure 5.3b.

When this refinement caused S to be split into S^- and S^+ and a is a deterministic action, we additionally do the following. We remove all outgoing edges from S^+ except the edge to T labeled with a. An illustration is shown in Figure 5.3d.

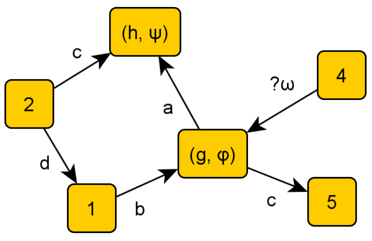
Listing 5.1 The abstraction refinement algorithm on D-PDL

THE ALGORITHM

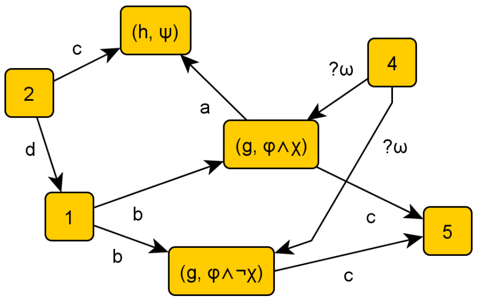
1. Initialize the abstraction A as the deterministic control flow graph of δ where each node is associated with the formula $\top$.
2. Split $(I, \top)$ into (I, φ_{pre}) and $(I, \neg\varphi_{pre})$. Split $(F, \top)$ into (F, φ_{post}) and $(F, \neg\varphi_{post})$.
3. Repeat the following
   (a) Find the shortest abstract path $S_0, \dots, S_n$ in A with $S_0 = (I, \varphi_{pre})$ and $S_n = (F, \neg\varphi_{post})$.
   (b) If such a path does not exist, output that δ is correct. Else, continue.
   (c) For $i \in \{0, \dots, n\}$ let ρ_i be the path constraint ρ_i of $S_0, \dots, S_i$. Find the smallest i such that ρ_i is not satisfiable.
   (d) If such an i does not exist, output that δ is incorrect. Continue otherwise.
   (e) Change the abstraction A by splitting along the path $S_0, \dots, S_i$.

EXPLANATION

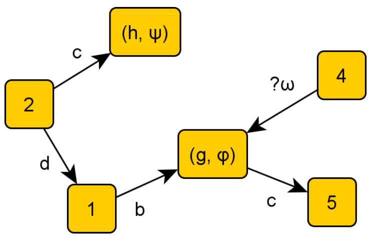
1. Abstractions and the initial abstraction are defined analogously to Definition 4.7 and 4.8 on page 24, where we use the deterministic control graph $G_{\mathrm{d}}(\delta)$ in stead of $G(\alpha)$.
2. In Definition 4.14 on page 26 it is defined how to split nodes.
3. In each iteration the abstraction A is changed, or the algorithm terminates.
   (a) An abstraction is a finite directed graph.
   (b) The correctness is proved in Theorem 5.17.
   (c) Path constraints are defined in Definition 4.12 on page 26.
   (d) The correctness is proved in Theorem 5.17.
   (e) Splitting along a path is defined in Definition 5.15 on page 41.



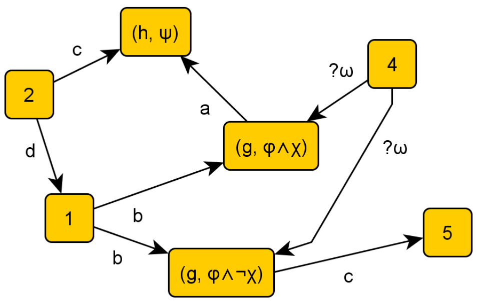
(a) Original abstraction



(b) When $\chi \wedge \varphi$ is satisfiable and a non-deterministic



(c) When $\chi \wedge \varphi$ is unsatisfiable



(d) When $\chi \wedge \varphi$ is satisfiable and a deterministic

Figure 5.3: Splitting the transition from (g, φ) to (h, ψ)

To prove the correctness of the algorithm, we want that abstractions are always overapproximations. The following Lemma shows that splitting along a transition using the new definition will result in an overapproximation.

Lemma 5.14. *Let A and A' be abstractions where A' is obtained from A by splitting along a transition according to Definition 5.13. If A is an overapproximation, then A' is an overapproximation.*

Proof. Let the transition from S to T with label a be the transition that is being split. Splitting along this transition is defined by first splitting according to Definition 4.16, let B be the resulting abstraction. By Lemma 4.17 on page 28 it follows that B is an overapproximation.

If S has not been split or a is non-deterministic, we have $A' = B$ and we are finished. Else, S has been split into S^- and S^+ and we have that A' equals B where certain edges are removed. We will prove that these edges cannot be part of a satisfying trace.

Let $S = (g, \varphi)$ and $T = (h, \psi)$, by Definition 4.16 we have $S^+ = (g, \varphi \wedge \chi)$ with $\chi = \mathrm{WP}_a(\psi)$. Let (s, r) be a (part of a) trace that satisfies the transition from S^+ to $T' = (h', \psi')$ with edge-label a'.

Since $s \vDash \varphi \wedge \chi$ we have by the definition of weakest preconditions that there exists a t with $(s, t) \in \mathcal{R}(a)$ and $t \vDash \psi$. By Lemma 5.10 on page 37 we have that the control flow graph $G_{\mathrm{d}}(\delta)$ is deterministic. Since a is the edge label of the edge from g to h with $(s, r) \in \mathcal{R}(a)$ and a' is the edge label of the edge from g' to h' with $(s, t) \in \mathcal{R}(a')$, we have by Definition 5.6 of deterministic control flow graphs that $a = a'$ and $h = h'$.

Since a is deterministic, we have $r = t$, so $r \vDash \psi$. Since ψ and ψ' are part of a partition of the state space and r satisfies both, we must have $\psi = \psi'$. Hence, $T = T'$.

The edge from S^+ to T with label a is not removed when splitting along the transition. Hence, A' is an overapproximation. □

We define splitting along a path in the same way as in Definition 4.18 on page 29, except that we use a different definition of splitting along a transition.

Definition 5.15 (Splitting along a path)**.** Let $S_0, \ldots, S_n$ be a path in the abstract program A with edge-labels $a_1, \ldots, a_n$. First we split A along the transition S_{n-1}, S_n with label a_n. If this split resulted in new states S^+ and S^-, and we have $n > 1$, then we recursively split along the path $S_0, \ldots, S_{n-1}, S^-$ with edge labels $a_1, \ldots, a_{n-1}$. Else we are finished.

Lemma 5.16. *Let A and A' be abstractions where A' is obtained from A by splitting along a path according to Definition 5.13. If A is an overapproximation, then A' is an overapproximation.*

Proof. Splitting along a path is defined by repeatedly splitting along a transition. The result follows from repeatedly applying Lemma 5.14. □

Correctness

Theorem 5.17. *If the algorithm terminates, its output is correct.*

Proof. The proof is the same as Theorem 4.26 on page 31, except that we use Lemma 5.16 to prove that the abstractions used are always overapproximations and we use Theorem 5.5 on page 35 for the equivalence between the denotational semantics and the structural operational semantics of deterministic propositional dynamic logic. □

5.4 Termination

The algorithm can fail to terminate for two reasons. One is that a check of the satisfiability of a formula does not terminate. The other is that the algorithm will explore infinite abstract counterexamples. Note that the halting problem can be translated to a verification question, so the algorithm cannot terminate on all inputs.

We assume that checking the satisfiability of formulas without modalities always terminates. Then we are able to prove that when all actions are deterministic the algorithm will halt on programs that do not satisfy their specification.

Lemma 5.18. *Assume that all actions $b \in \mathcal{B}$ are deterministic. Let A be an abstraction and A' be the resulting abstraction after splitting along a transition. Then there is an injective function from paths in A' ending at $(F, \neg\varphi_{post})$ to paths in A ending at $(F, \neg\varphi_{post})$ that preserves the length of the paths.*

Proof. See Definition 5.13 on page 38 for the definition of splitting along transitions, let S and T be the nodes of the transition and a the label of the edge. Since $(F, \neg\varphi_{post})$ has no outgoing edges, we have $S \neq (F, \neg\varphi_{post})$.

There are two possibilities. When the node S is not split, the difference between A and A' is that an edge has been removed. The identity serves as a length preserving injection.

Otherwise, the node S is split into S^- and S^+. We take the function that maps a path Γ to the path Γ' where all occurrences of S^- or S^+ are replaced with S. We have that S^- and S^+ cannot occur as the last node in the path, hence one if its outgoing edges is part of the path. By definition, the only outgoing edge from S^+ is the edge to T with label a and S^- does not have an edge with label a to T. Hence, this function is an injection. □

In the rest of this section, we call a path in A from (I, φ_{pre}) to $(F, \neg\varphi_{post})$ an error path.

Lemma 5.19. *Assume all actions $b \in \mathcal{B}$ are deterministic. Let A be an abstraction and let A' be the resulting abstraction after one iteration. Define n as the length of the smallest error path in A and m as the number of error paths of length n in A. Define n', m' similarly in A'. Then either $n' > n$; or $n' = n$ and $m' < m$.*

Proof. Since the algorithm considers the smallest path, the path it considers is of length n. The abstraction A' is obtained from A by splitting this path, which is defined as repeatedly splitting along a transition. For each of these transitions except the last, we invoke Lemma 5.18 to see that there are at most m error paths of length n. The last transition is being split without splitting a node, but only by removing an edge. This edge is part of the error path the algorithm considered. Hence, A' contains strictly less error paths of length n.

When A' does not contain any error path of length n anymore, we have $n' > n$. Otherwise, $n' = n$ and $m' < m$. $\square$

Theorem 5.20. *Assume all actions $b \in \mathcal{B}$ are deterministic. If the program δ is incorrect, the algorithm terminates.*

Proof. Suppose δ is incorrect. By Lemma 4.23 on page 30 A contains an error path of length n.

Let n_i be the length of the shortest error path during iteration i and m_i the number of paths with that length. If the algorithm does not terminate, we have for all i that $n_i \leq n$. Hence, there is a j with $n_{i+1} = n_i$ for all $i \geq j$. By Lemma 5.19 this gives an infinite decreasing sequence $(m_j, m_{j+1}, \ldots)$ of natural numbers which is a contradiction. We conclude that the algorithm terminates. $\square$

5.5 Verification of the lock/unlock example

Recall the program given in Listing 1.1 on page 3. We gave a translation to propositional dynamic logic in Listing 2.2 on page 10. The program uses only deterministic control flow statements, we can therefore translate it to deterministic PDL. The result is given in Listing 5.2 on the next page. The actions $has\_m := random$ and $foo := random$ are non-deterministic, all other actions are deterministic.

We will verify this program against the specification

$$\varphi_{pre} = \varphi_{post} = (\neg error \wedge \neg lock) .$$

Its initial abstraction is shown in Figure 5.4 on page 45. We have omitted the node $(I, \neg\varphi_{pre})$, since it is not used in the abstraction refinement algorithm. After 12 iterations the algorithm terminates and outputs that the program is correct. The final abstraction contains 55 nodes. The nodes that can be reached from (I, φ_{pre}) are shown in Figure 5.5 on page 46.

Recall that in the introduction we made the observation that has_m is true if and only if the program has the lock. This is exactly the formula that the algorithm found to distinguish between states that exit the loop and states that reenter the loop.

The final abstraction can serve as a certificate that the program is correct as follows. We can check whether actions that are removed are indeed not possible given the constraints in the abstract states and that the constraints are satisfied by all states at that point during the execution.

Listing 5.2 Lock/unlock program in D-DPL

```
has_m := ⊥;
while (¬has_m) {
    if (lock) {
        error := ⊤
    } else {
        lock := ⊤
    };
    has_m := random;
    if (has_m) {
        foo := random
    } else {
        if (lock) {
            lock := ⊥
        } else {
            error := ⊤
        };
        foo := random
    }
};
foo := random;
if (lock) {
    lock := ⊥
} else {
    error := ⊤
}
```

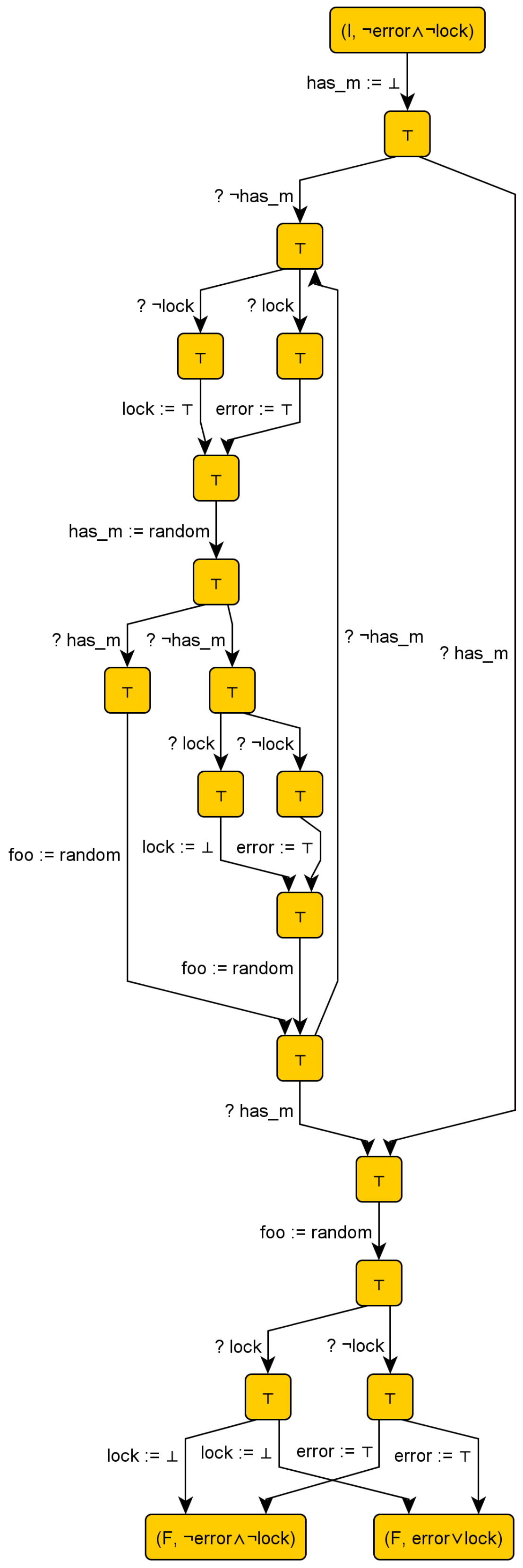


Figure 5.4: The initial abstraction

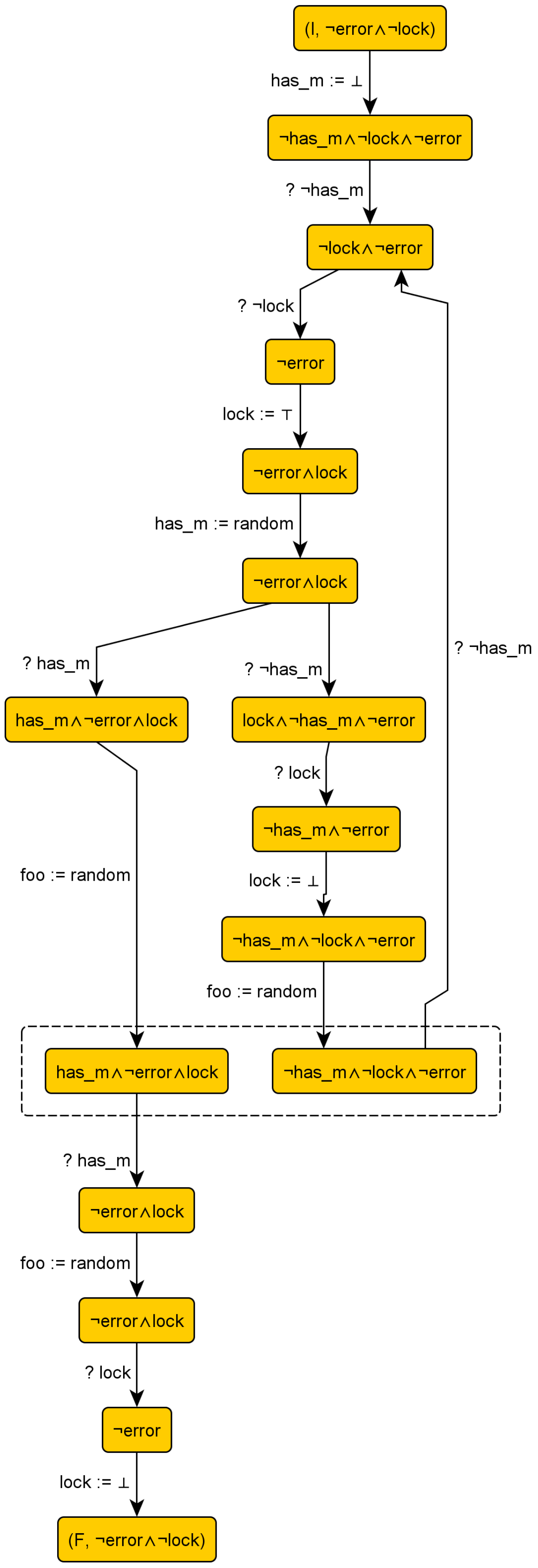


Figure 5.5: The final abstraction

Chapter 6

Indirection

An example of indirection in a programming language are pointers in C. The value of a variable that is a pointer is interpreted as an address. It can be dereferenced to obtain the value of the location with that address. There can be multiple layers of indirection when pointers point to an address that contains another address as a value. Furthermore, addresses can be manipulated by adding or subtracting numbers, this is called pointer arithmetic. See Figure 6.1 for an illustration, see [9] for an introduction to C and pointers.

By the nature of indirection, whether two pointers refer to the same location or not can greatly influence the execution of the program. This forces weakest preconditions to contain many case distinctions, which makes it infeasible to compute path constraints of large paths. As an alternative way of computing path constraints we introduce symbolic execution, see [11].

In Section 6.1 we introduce a language that has indirection. We show in Section 6.2 by an example that weakest preconditions cannot efficiently be used to compute path constraints in this language. We introduce symbolic execution in Section 6.3 as an alternative and prove the correctness in Section 6.4.

6.1 Language

Define the signature $(\mathcal{P}, \mathcal{B})$ as follows. We give propositions an internal structure by using terms.

Definition 6.1. The set of terms $\mathcal{T}$ is recursively defined, with $n \in \mathbb{Z}$, $v_1, v_2 \in \mathcal{T}$

$$v \quad ::= \quad n \mid \star v_1 \mid v_1 + v_2 \mid v_1 - v_2.$$

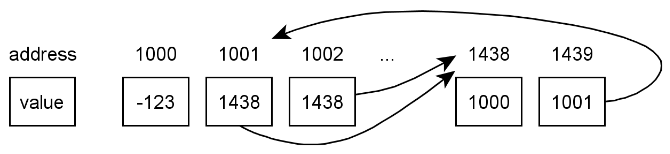


Figure 6.1: Memory with pointers

Then the set of propositions $\mathcal{P}$ is defined as follows, with $v_1, v_2 \in \mathcal{T}$

$$p ::= v_1 \simeq v_2 \mid v_1 > v_2.$$

Finally, the set of basic actions is

$$\mathcal{B} = \{\star v_1 := v_2 \mid v_1, v_2 \in \mathcal{T}\}.$$

The term $\star v$ is the term that introduces indirection, this will be clear when we have defined the semantics. The intended meaning of $v_1 \simeq v_2$ is that v_1 and v_2 are equal, to avoid confusion with other uses of $=$, we have chosen the symbol $\simeq$.

Definition 6.2. The model $M = (\Sigma, \mathcal{V}, \mathcal{R})$ is defined as follows. Let Σ be the set of functions $f : \mathbb{Z} \to \mathbb{Z}$. A state $s \in \Sigma$ determines a function $s_0^+ : \mathcal{T} \to \mathbb{Z}$ and a function $s_1^+ : \mathcal{P} \to \{\top, \bot\}$ as follows. Usually we omit the subscripts 1 and 2 and write s^+.

$$\begin{array}{rcl} s_0^+(n) & = & n \\ s_0^+(\star v) & = & s(s_0^+(v)) \\ s_0^+(v_1 + v_2) & = & s_0^+(v_1) + s_0^+(v_2) \\ s_0^+(v_1 - v_2) & = & s_0^+(v_1) - s_0^+(v_2) \end{array}$$

$$\begin{array}{rcl} s_1^+(\top) & = & \top \\ s_1^+(v_1 \simeq v_2) & = & \begin{cases} \top & \text{if } s_0^+(v_1) = s_0^+(v_2) \\ \bot & \text{if } s_0^+(v_1) \neq s_0^+(v_2) \end{cases} \\ s_1^+(v_1 > v_2) & = & \begin{cases} \top & \text{if } s_0^+(v_1) > s_0^+(v_2) \\ \bot & \text{if } s_0^+(v_1) \not> s_0^+(v_2) \end{cases} \\ s_1^+(\neg\varphi_1) & = & \neg s_1^+(\varphi_1) \\ s_1^+(\varphi_1 \wedge \varphi_2) & = & s_1^+(\varphi_1) \wedge s_1^+(\varphi_2). \end{array}$$

Then we define $\mathcal{V}(p) = \{s \mid s^+(p) = \top\}$. Let $n, m \in \mathbb{Z}$, define $s[n \mapsto m]$ as the function r with $r(n) = m$ and $r(k) = s(k)$ for all $k \neq n$. Finally the relations are defined by

$$\mathcal{R}(\star v_1 := v_2) \quad = \quad \{(s, r) \mid r = s[s(v_1) \mapsto s(v_2)]\}.$$

Note that all actions $b \in \mathcal{B}$ are deterministic.

6.2 Path constraints

Consider the program in Listing 6.1 on the following page, we want to check the postcondition $\star(-1) > \star 1$. This program will produce the same result for all starting states, so we can check it by hand and see that the postcondition is always satisfied.

However, this program is difficult to verify using the abstraction refinement algorithm. We will compute a part of the path constraint that leads the execution to a state where $\star(-1) \leq \star 1$. Recall that path constraints are inductively defined, starting at the last transition and using weakest preconditions.

Listing 6.1 A program with indirection

```
    ⋆-1 := 1;
    ⋆1 := -1;
    ⋆⋆1 := -⋆⋆1;
    ⋆⋆-1 := -⋆⋆-1;
    ⋆⋆-1 := -⋆⋆-1;
    ⋆⋆1 := -⋆⋆1;
```

Example. Let ρ_0 be $\neg\varphi_{post} = \star(-1) \leq \star 1$, we compute $\rho_i = \mathrm{WP}_{a_i}(\rho_{i-1})$ where a_i is the last but i-th action.

We have a_0 is $\star\star 1 := -\star\star 1$, to be able to substitute the new value of $\star\star 1$ we have to make a case distinction on $\star 1$. We see

$$\begin{aligned}
\rho_1 &= \mathrm{WP}_{a_0}(\rho_0) \\
&= (\star 1 \simeq 1 \to \rho_0[\star 1 \mapsto -\star 1]) \wedge (\star 1 \simeq -1 \to \rho_0[\star(-1) \mapsto -\star(-1)]) \\
&= (\star 1 \simeq 1 \to \star(-1) \leq -\star 1) \wedge (\star 1 \simeq -1 \to -\star(-1) \leq \star 1)\,.
\end{aligned}$$

To compute $\rho_2 = \mathrm{WP}_{a_1}(\rho_1)$ we have to make a new case distinction.

$$\begin{aligned}
\rho_2 &= \mathrm{WP}_{a_1}(\rho_1) \\
&= (\star(-1) \simeq 1 \to \rho_1[\star 1 \mapsto -\star 1]) \wedge (\star(-1) \simeq -1 \to \rho_1[\star(-1) \mapsto -\star(-1)]) \\
&= (\star(-1) \simeq 1 \to \star(-1) \leq \star 1) \\
&\quad \wedge (\star(-1) \simeq -1 \to (\star 1 \simeq 1 \to -\star(-1) \leq -\star 1) \wedge (\star 1 \simeq -1 \to \star(-1) \leq \star 1))\,.
\end{aligned}$$

Then we compute

$$\begin{aligned}
\rho_3 &= \mathrm{WP}_{a_2}(\rho_2) \\
&= (\star(-1) \simeq 1 \to \rho_2[\star 1 \mapsto -\star 1]) \wedge (\star(-1) \simeq -1 \to \rho_2[\star(-1) \mapsto -\star(-1)])\,.
\end{aligned}$$

Etcetera.

Computing path constraints using weakest preconditions quickly becomes infeasible. To overcome this problem we introduce a variant of symbolic execution.

6.3 Symbolic execution

Symbolic execution can be used to compute a path constraint. This works as follows. For each input value we introduce a symbol. An expression normally evaluates to a concrete value, now we symbolically evaluate an expression to a term over symbols. Assignments are executed symbolically by assigning terms over symbols to variables. For each control flow statement we obtain a boolean term over the symbols. The conjunction of these boolean terms forms the path constraint of the path.

We define the symbolic terms and formulas below. The variant of symbolic execution that we will define uses symbols to remove indirection from symbolic terms. Our symbolic language therefore does not contain the operator $\star$.

Definition 6.3. Let $\mathcal{X} = \{x_0, x_1, \ldots\}$ be a set of symbols. Define the set $\mathcal{T}^{\mathrm{X}}$ of symbolic terms recursively, with $x \in \mathcal{X}$, $n \in \mathbb{Z}$, $v_1^{\mathrm{x}}, v_2^{\mathrm{x}} \in \mathcal{T}^{\mathrm{X}}$

$$v^{\mathrm{x}} \quad ::= \quad x \mid n \mid v_1^{\mathrm{x}} + v_2^{\mathrm{x}} \mid v_1^{\mathrm{x}} - v_2^{\mathrm{x}}.$$

Define the set $\mathcal{P}^{\mathrm{X}}$ of symbolic propositions, with $v_1^{\mathrm{x}}, v_2^{\mathrm{x}} \in \mathcal{T}^{\mathrm{X}}$

$$p^{\mathrm{x}} ::= v_1^{\mathrm{x}} \simeq v_2^{\mathrm{x}} \mid v_1^{\mathrm{x}} > v_2^{\mathrm{x}}.$$

Finally, define the set Φ^{X} of symbolic formulas recursively, with $p^{\mathrm{x}} \in \mathcal{P}^{\mathrm{X}}$ and $\varphi_1^{\mathrm{x}}, \varphi_2^{\mathrm{x}} \in \Phi^{\mathrm{X}}$

$$\varphi^{\mathrm{x}} ::= \top \mid p^{\mathrm{x}} \mid \neg\varphi_1^{\mathrm{x}} \mid \varphi_1^{\mathrm{x}} \wedge \varphi_2^{\mathrm{x}}.$$

An assignment that maps symbols to concrete values connects a symbolic execution to a concrete execution.

Definition 6.4. An assignment is a function κ from $\mathcal{X}$ to $\mathbb{Z}$. Assignments can be extended to a function $\kappa_0 : \mathcal{T}^{\mathrm{X}} \to \mathbb{Z}$ and a function $\kappa_1 : \Phi^{\mathrm{X}} \to \{\top, \bot\}$ as follows. Usually we will omit the subscripts.

$$\begin{aligned}
\kappa_0(x) &= \kappa(x)\\
\kappa_0(n) &= n\\
\kappa_0(v_1^{\mathrm{x}} + v_2^{\mathrm{x}}) &= \kappa_0(v_1^{\mathrm{x}}) + \kappa_0(v_2^{\mathrm{x}})\\
\kappa_0(v_1^{\mathrm{x}} - v_2^{\mathrm{x}}) &= \kappa_0(v_1^{\mathrm{x}}) - \kappa_0(v_2^{\mathrm{x}})
\end{aligned}$$

$$\begin{aligned}
\kappa_1(\top) &= \top\\
\kappa_1(v_1^{\mathrm{x}} \simeq v_2^{\mathrm{x}}) &= \begin{cases} \top & \text{if } \kappa_0(v_1^{\mathrm{x}}) = \kappa_0(v_2^{\mathrm{x}}) \\ \bot & \text{if } \kappa_0(v_1^{\mathrm{x}}) \neq \kappa_0(v_2^{\mathrm{x}}) \end{cases}\\
\kappa_1(v_1^{\mathrm{x}} > v_2^{\mathrm{x}}) &= \begin{cases} \top & \text{if } \kappa_0(v_1^{\mathrm{x}}) > \kappa_0(v_2^{\mathrm{x}}) \\ \bot & \text{if } \kappa_0(v_1^{\mathrm{x}}) \not> \kappa_0(v_2^{\mathrm{x}}) \end{cases}\\
\kappa_1(\neg\varphi_1^{\mathrm{x}}) &= \neg\kappa_1(\varphi_1^{\mathrm{x}})\\
\kappa_1(\varphi_1^{\mathrm{x}} \wedge \varphi_2^{\mathrm{x}}) &= \kappa_1(\varphi_1^{\mathrm{x}}) \wedge \kappa_2(\varphi_2^{\mathrm{x}}).
\end{aligned}$$

When we cannot symbolically evaluate an expression with indirection using existing symbols, we will introduce a new symbol that represents this term. A mapping specifies which terms the symbols represent. Since new symbols are introduced one by one and we can only use symbols that are previously introduced, we use the following definition.

Definition 6.5. A mapping ξ is a (finite) partial function $\xi \subseteq \mathcal{X} \times \mathcal{T}^{\mathrm{X}}$ such that $\xi(x_i)$ does not contain the symbols x_j with $j \geq i$ and such that if $\xi(x_i)$ exists, then $\xi(x_j)$ exists for all $j < i$.

The intuition behind this definition is as follows. Suppose $(x_0, t^{\mathrm{x}}) \in \xi$. Since t^{x} cannot contain symbols x_i with $i \geq 0$ we have that $t^{\mathrm{x}} \in \mathcal{T}$. We say that x_0 represents the term $\star t^{\mathrm{x}} \in \mathcal{T}$. We can now inductively compute the terms $t_i \in \mathcal{T}$ that the symbols x_i represent as follows. Let $(x_i, t^{\mathrm{x}}) \in \xi$. We have that t^{x} only contains symbols x_j with $j < i$, replace each occurring symbol x_j by t_j to obtain a term $t' \in \mathcal{T}$. Then $t_i = \star t' \in \mathcal{T}$.

Listing 6.2 Symbolic execution

The input is a path Γ with $S_i = (g_i, \psi_i)$ and

$$\Gamma = S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} S_n.$$

The goal is to find the smallest i such that $\rho \wedge \eta$ is unsatisfiable, where ρ is the path constraint of the path $S_0, \ldots, S_i$ and η the alias information of this path.

1. Initialize the symbolic memory to $\mu = \emptyset$, the alias information to $\eta = \top$, the path constraint $\rho = \top$ and the symbolic mapping to $\xi = \emptyset$.
2. Let $S_0 = (g_0, \psi_0)$. Evaluate ψ_0 to ψ_0^{x} as defined in Listing 6.3 on the following page. Update the path constraint $\rho = \psi_0^{\mathrm{x}}$.
3. Iterate the following for $i = 1, \ldots, n$
   (a) When $a_i = ?\varphi$, do the following.
      i. Evaluate φ to φ^{x}, evaluate ψ_i to ψ_i^{x}.
      ii. Change the path constraint ρ to $\rho \wedge \varphi^{\mathrm{x}} \wedge \psi_i^{\mathrm{x}}$.
   (b) Else $a_i = (\star v_1 := v_2)$.
      i. Evaluate v_1 to v_1^{x} and then v_2 to v_2^{x}.
      ii. Compute the alias w^{x} of v_1^{x} as defined in Listing 6.4 on page 53.
      iii. There is a pair $(w^{\mathrm{x}}, u^{\mathrm{x}}) \in \mu$, change this pair to $(w^{\mathrm{x}}, v_2^{\mathrm{x}})$.
      iv. Evaluate ψ_i to ψ_i^{x} and change the path constraint ρ to $\rho \wedge \psi_i^{\mathrm{x}}$.
   (c) Check satisfiability of $\rho \wedge \eta$. When it is not satisfiable, output i. Else, continue.

When we symbolically evaluate a term with indirection, we cannot always determine whether we already have a symbol that represents this term or not, and we will make a choice. These choices will be specified in a formula $\eta \in \Phi^{\mathrm{X}}$ called the alias formula.

The result of a symbolic assignment is stored in the symbolic memory we define below. Since assignments are of the form $\star t := s$ with $t, s \in \mathcal{T}$, the symbolic memory maps symbolic terms to symbolic terms.

Definition 6.6. A symbolic memory μ is a (finite) partial function $\mu \subseteq \mathcal{T}^{\mathrm{X}} \times \mathcal{T}^{\mathrm{X}}$.

The algorithm that symbolically executes along a path Γ in an abstraction of a program is given in Listing 6.2.

6.4 Correctness

Symbolic execution is defined in an algorithmic way, we prove its correctness also algorithmically. We replay the symbolic execution and construct concrete states s_i that follow this execution. We want that a state s_i somehow matches the symbolic memory μ. We define this in the following way.

Listing 6.3 Symbolic evaluation

EVALUATING TERMS

The input is a (non-symbolic) term $v \in \mathcal{T}$ to evaluate, the current path constraint ρ, alias information η, symbolic memory μ and mapping ξ. The algorithm outputs a symbolic term in $\mathcal{T}^{\mathrm{X}}$ that is the evaluation of v, it can change η, μ and ξ, but not ρ.

The algorithm is a recursive algorithm. The most interesting case is the case $v = \star u$, the other cases are a straightforward extension.

1. Make the following case distinction.

   Case 1. $v = n$. Output n.

   Case 2. $v = \star u$. First symbolically evaluate u to u^{x}. Compute the alias w^{x} of u^{x} as defined in Listing 6.4 on the following page. Output $\mu(w^{\mathrm{x}})$.

   Case 3. $v = v_1 + v_2$. First evaluate v_1 to v_1^{x}. Note that this can change η, ξ and μ. Using the new values of these, evaluate v_2 to v_2^{x} and output $v_1^{\mathrm{x}} + v_2^{\mathrm{x}}$.

   Case 4. $v = v_1 - v_2$. First evaluate v_1 to v_1^{x}, then evaluate v_2 to v_2^{x}. Output $v_1^{\mathrm{x}} - v_2^{\mathrm{x}}$.

EVALUATING FORMULAS

The input is a (non-symbolic) formula $\varphi \in \Phi$ to evaluate, the current path constraint ρ, alias information η, symbolic memory μ and mapping ξ. The algorithm outputs a symbolic formula in Φ^{X} that is the evaluation of φ, it can change η, μ and ξ, but not ρ.

The algorithm is a recursive algorithm.

1. Make the following case distinction. All cases are straightforward extensions of the symbolic execution of terms.

   Case 1. $\varphi = \top$. Output $\top$.

   Case 2. $\varphi = v_1 \simeq v_2$. First evaluate v_1 to v_1^{x}. Note that this can change η, ξ and μ. Using the new values of these, evaluate v_2 to v_2^{x} and output $v_1^{\mathrm{x}} \simeq v_2^{\mathrm{x}}$.

   Case 3. $\varphi = v_1 > v_2$. First evaluate v_1 to v_1^{x}, then evaluate v_2 to v_2^{x}. Output $v_1^{\mathrm{x}} > v_2^{\mathrm{x}}$.

   Case 4. $\varphi = \neg\psi$. Evaluate ψ to ψ^{x} and output $\neg\psi^{\mathrm{x}}$.

   Case 5. $\varphi = \psi_1 \wedge \psi_2$. First evaluate ψ_1 to ψ_1^{x}, then evaluate ψ_2 to ψ_2^{x}. Output $\psi_1^{\mathrm{x}} \wedge \psi_2^{\mathrm{x}}$.

Listing 6.4 Computing and choosing an alias

As input there is a term $v^{\mathrm{x}} \in \mathcal{T}^{\mathrm{X}}$ to compute the alias of, the current path constraint ρ, the alias information η, the symbolic memory μ and the mapping ξ. The algorithm outputs a term in $\mathcal{T}^{\mathrm{X}}$ that is the alias of v^{x}, it can change η, μ and ξ, but not ρ.

1. Define $U = \left\{u^{\mathrm{x}} \in \mathcal{T}^{\mathrm{X}} \mid \exists w^{\mathrm{x}} \in \mathcal{T}^{\mathrm{X}}\ (u^{\mathrm{x}}, w^{\mathrm{x}}) \in \mu\right\}$. Define the formulas for all $u^{\mathrm{x}} \in U$ the formula

$$\psi_{u^{\mathrm{x}}} \quad = \quad (v^{\mathrm{x}} \simeq u^{\mathrm{x}}) \wedge \left(\bigwedge_{w^{\mathrm{x}} \in U, w^{\mathrm{x}} \neq u^{\mathrm{x}}} \neg (v^{\mathrm{x}} \simeq w^{\mathrm{x}}) \right).$$

   And define the formula

$$\psi_{none} = \left(\bigwedge_{w^{\mathrm{x}} \in U} \neg (v^{\mathrm{x}} \simeq w^{\mathrm{x}}) \right).$$

2. For each $\psi \in \{\psi_{u^{\mathrm{x}}} \mid u^{\mathrm{x}} \in U\} \cup \{\psi_{none}\}$ check whether $\psi \wedge \eta \wedge \rho$ is satisfiable.

3. If multiple of these formulas are satisfiable, choose one ψ arbitrarily and update η to $\eta \wedge \psi$. Else, there is exactly one satisfiable, let ψ be the corresponding formula.

4. If ψ is of the form $\psi_{u^{\mathrm{x}}}$, then output u^{x}.

5. Else $\psi = \psi_{none}$, let $x_i \in \mathcal{X}$ be the first symbol that is not present yet in ξ. Update ξ by adding the pair (x_i, v^{x}). Update μ by adding the pair (v^{x}, x_i) and output v^{x}.

Definition 6.7. Let κ be an assignment, recall that it uniquely extends to a function $\mathcal{T}^{\mathrm{X}} \to \mathbb{Z}$. We say that a state s agrees with a memory map μ if for all pairs $(v^{\mathrm{x}}, u^{\mathrm{x}}) \in \mu$ we have $s\,(\kappa(v^{\mathrm{x}})) = \kappa(u^{\mathrm{x}})$.

Theorem 6.8. *Let A be an abstraction of a program α. Let Γ be a path in A starting at (I, φ_{pre}) and ending in $(F, \neg\varphi_{post})$ that is symbolically executed and suppose that the final constraint $\rho \wedge \eta$ is satisfiable. Then $M \nvDash \varphi_{pre} \to [\alpha]\varphi_{post}$.*

Proof. The proof is given in Listing 6.5 on the next page. $\square$

Listing 6.5 The replay of a symbolic execution

We will replay the symbolic execution of the path Γ with $S_i = (g_i, \psi_i)$ and

$$\Gamma = S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} S_n.$$

See Listing 6.2 on page 51 for the definition of symbolic execution. We assume that $\rho_i \wedge \eta_i$ is satisfiable at each iteration of the symbolic execution. Let κ be a satisfying assignment of the final value $\rho_n \wedge \eta_n$.

We will construct a trace $(s_0, \ldots, s_n)$ with $s_i \vDash \psi_i$ and $(s_{i-1}, s_i) \in \mathcal{R}(a_i)$ that proves that

$$s_0 \vDash \psi_0 \wedge \langle \alpha \rangle \psi_n.$$

At every stage we want that the current s_i agrees with the current memory map μ, see Definition 6.7 on the previous page. When a term $v \in \mathcal{T}$ or a formula $\varphi \in \Phi$ is symbolically evaluated, we have by Listing 6.6 on the following page that $s_i^+(v) = \kappa(v^{\mathrm{x}})$ and $s_i^+(\varphi) = \kappa(\varphi^{\mathrm{x}})$. The symbolic memory could change during the symbolic evaluation, but the algorithm in Listing 6.6 ensures that s_i still agrees with the resulting memory.

1. Define s_0 as the state with $s_0\left(\kappa(v_i^{\mathrm{x}})\right) = \kappa(x_i)$ for each $(x_i, v_i^{\mathrm{x}}) \in \xi_n$. We have that s_0 agrees with the current memory μ, since $\mu = \emptyset$.

2. The path constraint is updated to $\rho = \psi_0^{\mathrm{x}}$. Since ψ_0^{x} is a conjunct in the final path constraint ρ_n, we have that κ satisfies ψ_0^{x}. Since s_0 agrees with μ we have that $s_0 \vDash \psi_0$.

3. Iterate the following for $i = 1, \ldots, n$. We assume we have a s_{i-1} that satisfies the memory μ at this point. We construct a s_i that satisfies the memory at the end of this iteration, that has $s_i \vDash \psi_i$ and $(s_{i-1}, s_i) \in \mathcal{R}(a_i)$.

   (a) When $a_i =?\varphi$, do the following.

      i. The path constraint is updated to $\rho \wedge \varphi^{\mathrm{x}} \wedge \psi_i^{\mathrm{x}}$. Since these are conjuncts of the final path constraint ρ_n, we have that κ satisfies $\varphi^{\mathrm{x}} \wedge \psi_i^{\mathrm{x}}$.

      ii. Define $s_i = s_{i-1}$. Since s_{i-1} agrees with μ we have $s_i \vDash \psi_i$ and $s_{i-1} \vDash \varphi$ so $(s_{i-1}, s_i) \in \mathcal{R}(?\varphi)$.

   (b) Else $a_i = (\star v_1 := v_2)$.

      i. Let v_1^{x} and v_2^{x} denote the evaluations of v_1 and v_2. We have $s_{i-1}^+(v_1) = \kappa(v_1^{\mathrm{x}})$ and $s_{i-1}^+(v_2) = \kappa(v_2^{\mathrm{x}})$ since s_{i-1} agrees with μ.

      ii. Let w^{x} be the alias of v_1^{x} as defined in Listing 6.7 on the next page. We have $\kappa(w^{\mathrm{x}}) = \kappa(v_1^{\mathrm{x}})$ as proved in the same Listing.

      iii. The memory is updated by changing the pair $(w^{\mathrm{x}}, u^{\mathrm{x}}) \in \mu$ to the pair $(w^{\mathrm{x}}, v_2^{\mathrm{x}})$. Define s_i the same as s_{i-1}, except that $s_i\left(\kappa(w^{\mathrm{x}})\right) = \kappa(v_2^{\mathrm{x}})$. Since s_{i-1} agrees with the memory before this update, we have that s_i agrees with the memory after the update. Since $\kappa(w^{\mathrm{x}}) = s_{i-1}^+(v_1)$ and $\kappa(v_2^{\mathrm{x}}) = s_{i-1}^+(v_2)$ we have $(s_{i-1}, s_i) \in \mathcal{R}(\star v_1 := v_2)$.

      iv. The path constraint is updated to $\rho \wedge \psi_i^{\mathrm{x}}$. Since this is a conjunct of the final path constraint ρ_n, we have that κ satisfies ψ_i^{x}. Hence, $s_i \vDash \psi_i$.

Listing 6.6 The replay of a symbolic evaluation

Let s_i be a state in the replay of a symbolic execution that agrees with the memory μ at the stage that a term v or formula φ is being symbolically evaluated under the assignment κ. When a term v is symbolically evaluated to v^{x} we prove that $s_i^+(v) = \kappa(v^{\mathrm{x}})$. When a formula φ is symbolically evaluated to φ^{x} we prove that $s_i^+(\varphi) = \kappa(\varphi^{\mathrm{x}})$. Furthermore, we have that s_i still agrees with the memory after the symbolic execution.
The most interesting case is $v = \star u$.

- First u is symbolically evaluated to u^{x}. By induction hypothesis we have $\kappa(u^{\mathrm{x}}) = s_i^+(u)$.
- Then the alias w^{x} of u^{x} is computed, such that $\mu(w^{\mathrm{x}})$ is defined. By Listing 6.7 we have that $\kappa(w^{\mathrm{x}}) = \kappa(u^{\mathrm{x}})$.
- The symbolic evaluation of v is the $v^{\mathrm{x}} = \mu(w^{\mathrm{x}})$. Since s_i agrees with μ, we have $s_i(\kappa(w^{\mathrm{x}})) = \kappa(v^{\mathrm{x}})$.
- By definition we have $s_i^+(v) = s_i^+(\star u) = s_i(s_i^+(u))$. From $s_i^+(u) = \kappa(u^{\mathrm{x}}) = \kappa(w^{\mathrm{x}})$ it follows that $s_i^+(v) = s_i(\kappa(w^{\mathrm{x}})) = \kappa(v^{\mathrm{x}})$.

The other cases follow from recursion, since symbolic execution and the evaluation $s^+ : \mathcal{T} \to \mathbb{Z}$ follow the same structure.

Listing 6.7 Replaying the computation of an alias

We replay the computation of the alias w^{x} of the term $v^{\mathrm{x}} \in \mathcal{T}^{\mathrm{X}}$. We will prove that we have $\kappa(w^{\mathrm{x}}) = \kappa(v^{\mathrm{x}})$ and when s_i agrees with the memory, it will still agree with the memory after choosing the alias.
Let $U = \{u^{\mathrm{x}} \in \mathcal{T}^{\mathrm{X}} \mid \mu(u^{\mathrm{x}}) \text{ is defined}\}$ and $\Psi \in \{\psi_{u^{\mathrm{x}}} \mid u^{\mathrm{x}} \in U\} \cup \{\psi_{none}\}$ be as defined in Listing 6.4 on page 53.
Suppose the alias w^{x} was chosen using $\psi_{w^{\mathrm{x}}}$, see step 4 of Listing 6.4 on page 53. The memory is not updated, so s_i still agrees with the resulted memory. When there was one $\psi \in \Psi$ such that $\psi \wedge \eta \wedge \rho$ is satisfiable, we have that $v^{\mathrm{x}} \simeq w^{\mathrm{x}}$ follows from $\eta \wedge \rho$. Since κ satisfies $\rho_n \wedge \eta_n$ we have that $\kappa(v^{\mathrm{x}}) = \kappa(w^{\mathrm{x}})$. When there were multiple $\psi \in \Psi$, the constraint $v^{\mathrm{x}} \simeq w^{\mathrm{x}}$ is a conjunct in η_n, hence also $\kappa(v^{\mathrm{x}}) = \kappa(w^{\mathrm{x}})$.
Suppose the alias w^{x} was chosen using ψ_{none}, see step 5 of Listing 6.4 on page 53. Then $w^{\mathrm{x}} = v^{\mathrm{x}}$ so $\kappa(w^{\mathrm{x}}) = \kappa(v^{\mathrm{x}})$.
The memory is updated by choosing a new symbol $x_i \in \mathcal{X}$, adding the pair (x_i, v^{x}) to ξ and the pair (v^{x}, x_i) to μ. The pair (x_i, v^{x}) is part of ξ and therefore of ξ_n. By construction of s_0 (see Listing 6.5) we have $s_0\left(\kappa(v_i^{\mathrm{x}})\right) = \kappa(x_i)$.
Since ψ_{none} was chosen, we know that terms that could alias with v^{x} have not been evaluated before. In particular, assignments that change the value of v^{x} have not been encountered. Hence, $s_i\left(\kappa(v_i^{\mathrm{x}})\right) = s_0\left(\kappa(v_i^{\mathrm{x}})\right)$. We conclude that s_i also agrees with the new pair (v^{x}, x_i) in the symbolic memory.

Chapter 7

Conclusion

We defined programs and partial correctness specifications using propositional dynamic logic. To describe the individual steps that are part of executing a program, we introduced control flow graphs and defined a translation from PDL programs to control flow graphs. Then we defined an algorithm that can automatically verify PDL programs against partial correctness specifications. The algorithm works by counterexample-driven abstraction refinement.

We considered a deterministic fragment of propositional dynamic logic, we adapted the verification algorithm to improve its efficiency on deterministic programs and we proved that this algorithm terminates on incorrect programs. We ran this algorithm on the example program given in the introduction. The algorithm proved the correctness of this program by discovering the same invariant we gave as an argument to show the correctness of the program.

Finally, we introduced a language with indirection and we explained why using weakest preconditions is a very inefficient way of computing path constraints. We proposed symbolic execution as an alternative way to compute path constraints. Our variant of symbolic execution furthermore removes all indirection from the path constraint, hence indirection does not play a role when checking the path constraint for satisfiability.

As future research we would like to investigate how symbolic execution can be used to give an alternative definition of splitting along a path. The current definition uses weakest preconditions and suffers from the same inefficiency as computing path constraints with weakest preconditions. An idea would be to use the alias information to reduce the number of disjoints in the weakest precondition, the problem that should be overcome with this approach is that during the symbolic execution choices are made about which terms alias each other.

We believe that the termination result we proved for deterministic programs also holds for non-deterministic programs. Since splitting along a path in the non-deterministic case can introduce new error paths of the same length, the argument does not easily translate. An idea would be to group error paths in the abstraction by the path they represent in the control flow graph and show that the number of paths in one group strictly decreases.

Another direction for future research is to include the support for total correctness specifications. This would probably require a change of framework, since in propositional dynamic logic we cannot directly state that a program terminates. A connection with temporal logic could enable us to do so.

When we verified the example program, we observed that the postcondition $\neg error$ was propagated through the whole control flow graph. It could be more efficient to include an explicit way of exiting the program with an error and adapt the abstraction refinement algorithm to search for paths that end in a state where an error is thrown.

Finally, we assumed that we have an efficient way of deciding the satisfiability of formulas without modalities. In practice this will be the bottleneck of the algorithm, some approaches run concrete executions of the program in parallel and use information of this execution to simplify the path constraint. It would be interesting to investigate if including information of a concrete execution still enables the algorithm to prove that a program is correct.

Bibliography

[1] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, Sai Deep Tetali, and Aditya V. Thakur. Proofs from tests. *IEEE Transactions on Software Engineering*, 36(4):495–508, July-August 2010.

[2] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2002.

[3] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[5] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[7] Michael J. Fischer and Richard E. Ladner. Propositional modal logic of programs. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 286–294, New York, NY, USA, 1977. ACM.

[8] R. W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. The American Mathematical Society, 1967.

[9] David Griffiths and Dawn Griffiths. *Head First C*. O'Reilly Media, Inc., 2012.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[11] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John

Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619, pages 553–568. Springer Berlin / Heidelberg, 2003.

[12] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications - A Formal Introduction.* John Wiley & Sons, 1999.

[13] Rohit Parikh. The completeness of propositional dynamic logic. In J. Winkowski, editor, *Mathematical Foundations of Computer Science 1978*, volume 64 of *Lecture Notes in Computer Science*, pages 403–415. Springer Berlin / Heidelberg, 1978.

[14] Vaughan R. Pratt. Semantical consideration on Floyd-Hoare logic. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 109 –121, oct. 1976.

[15] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. *ESEC-FSE*, pages 263–272, September 2005.

[16] Jan van Eijck and Martin Stokhof. The gamut of dynamic logics. In Dov Gabbay and John Woods, editors, *Handbook of the History of Logic*, volume 7, pages 499–600. Elsevier, 2006.