Universität Stuttgart
Institut für Formale Methoden der Informatik
Abteilung Sichere und Zuverlässige Softwaresysteme
Fakultät Informatik, Elektrotechnik und Informationstechnik
Universitätsstr. 38, 70569 Stuttgart

Diplomarbeit No. 2284

Abstraction Refinement for Pushdown Systems

Stefan Kiefer

Course of study:	Informatik
Examiner:	Prof. Dr. Javier Esparza
Supervision:	Dr. Stefan Schwoon
Started:	25.11.2004
Finished:	24.05.2005
CR Classification:	D.2.4, F.3.1

Declaration

I hereby declare that this thesis was composed by myself, and that I have used no sources other than those cited in the thesis.

_____

(Stefan Kiefer)

Zusammenfassung

Diese Arbeit passt das Paradigma der gegenbeispiel-geleiteten Abstraktionsverfeinerung auf das Model-Checken von Kellerautomaten an.

Es wird eine auf Craig-Interpolation basierende Theorie entwickelt, die anschließend auf ein Abstraktionverfeinerungs-Schema für sequentielle Programme angewandt wird. Diese Konzepte werden so verallgemeinert, dass Kellerautomaten, einschließlich Rekursion, sowie mehrfache Gegenbeispiele behandelt werden können.

Diese Grundlagen bereiten den Weg für verschiedene Heuristiken, die zur Berechnung aussagekräftiger Prädikate herangezogen werden können. Mehrere konkrete Heuristiken werden vorgeschlagen und diskutiert.

Eine Implementierung, die auf dem Model-Checker Moped basiert, zeigt die Nützlichkeit der entwickelten Konzepte. Im Gegensatz zu anderen Ansätzen werden durchgängig binäre Entscheidungs-Diagramme (BDDs) verwendet.

Abstract

This thesis adapts the paradigm of CEGAR (counterexample-guided abstraction refinement) to the model checking of pushdown systems.

A theoretical framework based on Craig interpolation is developed and applied to the automatic abstraction of sequential programs. It is generalized to handle full pushdown systems, including recursion, as well as multiple counterexamples.

It is shown that this theory provides a framework for different heuristics to compute relevant predicates. Several concrete heuristics are proposed and discussed.

An implementation based on the model checker Moped gives evidence of the usefulness of the developed concepts. In contrast to other approaches, Binary Decision Diagrams (BDDs) are used throughout the CEGAR loop.

Acknowledgements

My foremost words of thanks go to Javier Esparza for all his commitment. He introduced model checking to me in a very inspiring way and generously offered me to join his group. Thanks for your great trust in me and all your time, friendliness, help and inspiration!

I am equally thankful for Stefan Schwoon's exemplary supervision. He worked hard to make this work possible, and our cooperation could not have been more friendly or more effective. Thanks for all your time and much helpful advice!

I thank my other colleagues at our group, who all welcomed me in such an open and friendly way. Special thanks to Dejvuth Suwimonteerabuth – Remy, your smiles and your work in our "software production office" helped a lot! I am looking forward to the coming years in this group.

Many people have supported me during my university education. It is Volker Diekert at the Universität Stuttgart who made a lot possible. At Michigan Tech, I owe thanks especially to Linda M. Ott and Charles Wallace. I cannot mention all the friends and teachers who helped me during my education, thanks to all of you!

Unending thanks to my parents for their constant love, encouragement and support in all the years — I owe them everything.

Contents

Chapter 1

Introduction

1.1 Formal verification

Computers play an increasingly important role in our society. More and more tasks in both professional and private domains are assigned to computers.

On the one hand, the processor speed has been increasing for many years, which, for instance, helps us tackle computationally expensive problems, particularly in the area of simulation and scientific computing.

On the other hand, the foreword of [CGP02] says: "It is widely agreed that the main obstacle to 'help computers help us more' and relegate to these helpful partners even more complex and sensitive tasks is not inadequate speed and unsatisfactory raw computing power in the existing machines, but our limited ability to design and implement complex systems with sufficiently high degree of confidence in their correctness under all circumstances."

This problem is evident especially in safety-critical systems, e. g., in airplanes or medical devices, where a computer failure has to be avoided by all means.

The traditionally practiced methods for increasing the confidence in a computer system are *simulation and testing*. Whereas those techniques are indispensable in earlier stages of the system, they get less and less effective, when the most obvious errors have been removed. In most cases, one cannot test all cases, because this would take too much time or there are infinitely many cases to test. Therefore, if testing is the only validation technique used, then a risk is taken that some errors are not found and therefore remain in the system.

Formal verification, in contrast, aims at a formal proof that a system fulfills certain properties, e. g., does not contain particular errors. A formal

description of the model and the properties to be checked are needed. Using mathematical methods, a proof for the correctness of the system is sought.

1.2 Software model checking

Model checking is one instance of a formal verification method. The foreword of [CGP02] lists two particularly attractive advantages of this technique:

- "It is fully automatic, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving. Anyone who can run simulations of a design is fully qualified and capable of model-checking the same design. In the context of currently practiced techniques, model checking can be viewed as the ultimately superior simulation tool.

- When the design fails to satisfy a desired property, the process of model checking always produces a *counterexample* that demonstrates a behavior which falsifies the property. This faulty trace provides a priceless insight to understanding the real reason for the failure as well as important clues for fixing the problem."

While originally designed for the verification of hardware, *software model checking* is a promising approach also for improving software reliability and robustness.

The formal description of a software system is just the program code or a model of it. The specification of the properties to be checked is usually given in terms of a formula of some temporal logics, e. g., LTL (Linear Temporal Logic). In the simplest case, it is the reachability or non-reachability of some label in the program code.

1.3 The state explosion problem

The main obstacle of (software) model checking is the *state explosion problem*. Simply put, the problem is that the program can assume many states and thus render the proof of its correctness difficult.

A simple example is a program with n boolean variables. Each such variable can be set to 0 or 1, i. e., the program has 2^n states. Here, the number of states might be exponentially bigger than the size of the program code.

There are several approaches to tackle this problem. They can be classified into three main categories:

1. Compression. This concerns the design of efficient data structures that are able to represent large sets of states. A classical example for such a state representation are Binary Decision Diagrams [Bry86].

2. Reduction. [Sch02] characterizes this as "cutting away states and execution sequences without affecting the validity of the property in question. For instance, if a system can execute one or more steps in any order, but the order in which the steps happen has no influence on whether the property holds or not, then nothing is gained by exploring all possible orders, and one can reduce the system to one where some arbitrary order is fixed."

3. Abstraction. Abstraction techniques consider only some aspects of the system and thus make it simpler. Many states might effectively be merged to a single one, because they cannot be distinguished by looking only at those aspects. We describe abstraction in more detail in Section 1.4.

1.4 Model checking and abstraction

The general idea behind abstraction in model checking is to reduce the state space by looking only at some aspects of the system to be model-checked.

Example 1.1. Consider the following simple program:

```
X := 1
for Y := 1 to 1000 do
     for Z := 1 to 1000 do
          if Y = Z · Z then print Y
W := X + 1
X := X/W
```

We want to model-check the fact that the last line does not cause a *division-by-zero* error. Although this program has many reachable states, it is easy to see that the variable W will never be 0. To verify this, one need not understand in detail what the nested loops do. It suffices to notice that the nested loops cannot influence the value of W. In fact, we can ignore the variables Y and Z. In other words, we can look at an *abstraction* of the above program:

```
X := 1
skip
skip
skip
W := X + 1
X := X/W
```

This program's state space is considerably smaller and probably faster to model-check.

The method used in this example is called *slicing*: Compute the variables that have a potential influence on the property to be checked. Then, abstract the program by removing all statements that do not modify variables with a potential influence. This method can be of great benefit for model checking [DH99].

However, there are also other methods for systematic abstraction of programs. In particular, *predicate abstraction* [GS97] is widely used. It abstracts data by only keeping track of certain predicates over the original variables. Each (boolean) variable in the abstract program corresponds to some predicate over the original program variables. The original variables are eliminated. We describe predicate abstraction in our context in Section 4.1.

In the remainder of this work, we focus on predicate abstraction as our abstraction technique. It can be used in abstraction refinement schemes, as described in the next section.

1.5 Counterexample-guided abstraction refinement

When talking about abstraction in software model checking, one should clarify the relation between the abstract program and the original concrete program. Typically, one requires that the abstract program be an *over-approximation* of the concrete program, i. e., it admit "more" behaviors than the concrete program. However, those additional behaviors might include "bad" behaviors, i. e., behaviors that violate the specification. As a consequence we might not be able to verify the property in the specification.

One idea is to (automatically) detect such cases, i. e., a bad behavior that is due to abstraction and not present in the original program. Those cases of "over-abstraction" might guide us in revising our abstraction, i. e., in an *abstraction refinement*.

To describe this abstraction refinement idea more precisely, we assume that we already have some abstract program. Model-checking this abstract program can yield one out of two results:

- The abstract program does not admit any bad behavior. In this case, it can be concluded that the concrete program cannot show any bad behavior either. In other words, it fulfills the specification.
- The abstract program admits some bad behavior. Usually the model checker illustrates this by means of a *counterexample*. In this case, there are again two possibilities:
  - The counterexample is *spurious*, i. e., it results from the over-approximation and does not occur in the concrete program.
  - The counterexample is *real*, i. e., the bad behavior also occurs in the concrete program. In this case, it can be concluded that the concrete system violates the specification.

In the case of a spurious counterexample, this counterexample can be inspected in order to extract predicates to be added to the abstraction. The new predicates should be chosen such in a way that the spurious behavior is ruled out in the new refined abstraction.

This is called the CEGAR paradigm (counterexample-guided abstraction refinement) and was introduced in [CGJ+00]: Start with a coarse abstraction (e. g., no data, only control flow) and use the spurious counterexample for refining the abstraction. Predicate refinement is repeated until the abstract program no longer contains bad behaviors or the model checker delivers a real counterexample. Figure 1.1 (borrowed from [CKSY04]) illustrates this process. All steps can be carried out automatically.

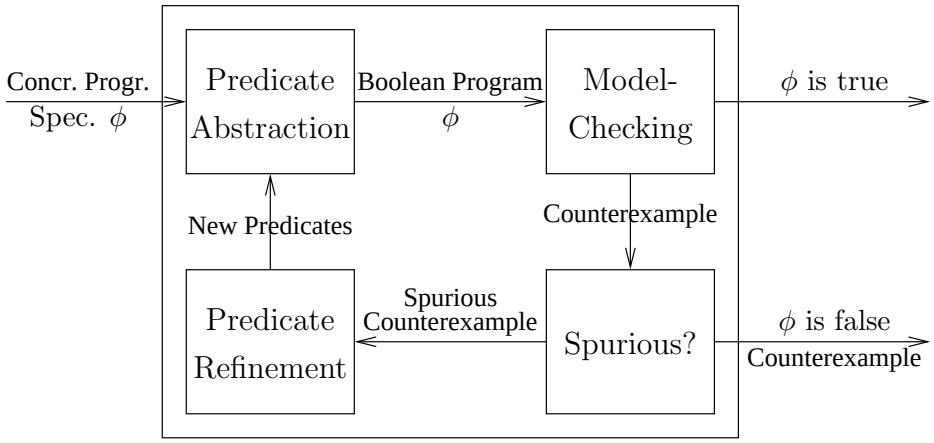

Figure 1.1: The CEGAR Loop.

Counterexample-guided abstraction refinement (CEGAR) is a powerful tool for automated abstraction of hardware and software systems. Originally designed for verification of hardware designs, this technique was successfully utilized for software verification as well. Particularly, the SLAM project [BR01] has gained attention and has demonstrated the effectiveness of software verification for device drivers. The BLAST tool [HJMS02] and the MAGIC tool [CCG+03] have been applied successfully in domains of security protocols and real-time operating-system kernels.

1.6 Organization of this thesis

The remainder of this thesis is organized as follows:

- In Chapter 2 pushdown systems are defined, which are our model for the systems to be checked. Moped, a model checker for pushdown systems, is presented. At the end of this section, we compare our work with related work.
- In Chapter 3 we establish a theoretical framework based on Craig interpolation. We will base our predicate-generation scheme on this theory.
- Chapter 4 describes our approach for a CEGAR scheme.
- In Chapter 5 this approach is generalized to full pushdown systems.
- In Chapter 6 we suggest and discuss several heuristics along with experiments motivating them.
- Chapter 7 deals with some aspects regarding the efficiency of our implementation.
- In the Chapters 8 and 9, we suggest future work and provide conclusions of this thesis.

Chapter 2

Pushdown systems

In model checking, one needs a model of the system to be checked. The behavior of sequential programs with procedures (including possible recursion) can be naturally modeled with pushdown systems.

This section is organized as follows. In Section 2.1 we define basic notions of pushdown systems. In Section 2.2 symbolic pushdown systems, a specialization of pushdown systems, are introduced. Section 2.3 presents Moped, a model checker for symbolic pushdown systems. In Section 2.4 we sketch our work, comparing it with related work.

2.1 Definitions

Definition 2.1 (Pushdown system)**.**
A pushdown system (PDS) is a quadruple $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ *with the following properties:*

- P *is a finite set of* control states*.*
- Γ *is a finite set of* stack symbols*.*
- $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ *is a finite set of* pushdown rules*. If* $((p, \gamma), (p', w)) \in \Delta$*, we also write* $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$*.*
- c_0 *is an* initial configuration*.*

A configuration *is an element of* $P \times \Gamma^*$*.*

A PDS defines a relation between configurations:

Definition 2.2 (Pushdown transition)**.**
If $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$*, then* $\langle p, \gamma w' \rangle \Rightarrow_{\mathcal{P}} \langle p', ww' \rangle$ *for all* $w' \in \Gamma^*$*.*

In other words, a transition modifies a configuration in the following way: It can change the control state and replace the "topmost" (i. e. leftmost) stack symbol by a string of other symbols including the empty string (effectively removing the topmost symbol). The symbols in the stack below the topmost stack symbol remain untouched.

Definition 2.3 (Computation of a pushdown system)**.** *A* computation *of a PDS is a sequence of configurations of the following form:*

$$c_0 \Rightarrow_{\mathcal{P}} c_1 \Rightarrow_{\mathcal{P}} c_2 \Rightarrow_{\mathcal{P}} \cdots,$$

where c_0 is the initial configuration.

Computations of PDSs model the behavior of sequential programs with procedures. When talking about *model checking a pushdown system*, one wants to verify properties that hold for the computations of the given PDS. Even though there might be infinitely many reachable configurations, one can efficiently model-check PDSs for short specifications [BEM97]:

Theorem 2.4. *The model checking problem for LTL and pushdown systems is DEXPTIME-complete, and polynomial in the size of the PDS for a fixed formula.*

2.2 Symbolic pushdown systems

Sequential programs usually feature local and global variables. Local variables could be encoded in the stack symbols; global variables in the control state. For simplicity, we assume that we have n_G many global boolean variables and, for each procedure, n_L many local boolean variables.[1] The following restricted form of a PDS includes variables explicitly.

Definition 2.5 (Symbolic pushdown system[2] [3])**.**
A symbolic pushdown system *is a pushdown system of the form $(G, \Gamma_0 \times L, \Delta, C_0)$, where*

- *G is the set $\{0,1\}^{n_G}$ of global variable evaluations,*
- *Γ_0 is a set of procedure* control points*,*
- *L is the set $\{0,1\}^{n_L}$ of local variable evaluations,*

[1] Integer variables with a finite range can be encoded as booleans.
[2] This definition is slightly more restrictive than in [Sch02].
[3] More recently, symbolic PDSs have been regarded as *weighted* PDSs [RSJMar].

- Δ *is a set of* symbolic transition rules*, where each rule is of the form*

$$\langle \gamma \rangle \hookrightarrow \langle \gamma_1 \dots \gamma_n \rangle \quad (R)$$

with

  - $n \leq 2$
  - $\gamma, \gamma_1, \dots, \gamma_n \in \Gamma_0$
  - $R \subseteq (G \times L) \times (G \times L^n)$,

- $C_0 = G \times (\{y_0\} \times L)$ *is the set of initial configurations.*

Notice that we slightly deviate from a pushdown system in this definition, since we may have multiple initial configurations. However, the set C_0 is already given by the initial control point γ_0. This control point is the "start address" of the program. The form of C_0 implies that no assumptions are made about the initial variable values.

The right side of the rules can consist of 0, 1 or 2 control points.

- A rule with 1 control point on the right side describes a *normal* statement: By executing it, the current control point is transformed into a new control point and the local and global variables are transformed according to the relation R. Here, R is of the form $R(g_1, \dots, g_{n_G}, l_1, \dots, l_{n_L}, g'_1, \dots, g'_{n_G}, l'_1, \dots, l'_{n_L})$, where the non-primed variables refer to the global and local variables before execution of the statement, and the primed variables refer to the variables after execution.

- A rule with 2 control points on the right side describes a *push* statement, which models a procedure call: By executing it, the current control point is transformed into two new control points: the top (left) control point is the entry point where the newly called procedure starts. The other control point is a return address where execution should be resumed after returning from the newly called procedure. The relation R can restrict the global variables as well as the local variables of the current and the called procedure. More precisely, R is of the form $R(g_1, \dots, g_{n_G}, l_1, \dots, l_{n_L}, g'_1, \dots, g'_{n_G}, l'_1, \dots, l'_{n_L}, l''_1, \dots, l''_{n_L})$, where

  - the non-primed variables refer to the global and local variables before execution of the statement,
  - the primed global variables refer to the global variables after execution,
  - the primed local variables refer to the initial values of the variables local to the called procedure,

  - the double-primed local variables refer to the variables local to the procedure that is resumed after returning from the procedure call.

- A rule with 0 control points on the right side describes a *pop* statement, which models a return from a procedure: By executing it, the current procedure terminates itself. Execution will continue from the return address specified when the procedure was once called. The relation R can return values by restricting the global variables. More precisely, R is of the form $R(g_1, \ldots, g_{n_G}, l_1, \ldots, l_{n_L}, g'_1, \ldots, g'_{n_G})$, where the non-primed variables refer to the global and local variables before execution of the statement, and the primed global variables refer to the global variables after execution.

The representation of PDSs as *symbolic* PDSs has two main advantages:

- There is a natural correspondence between procedural programs and PDSs.
- The part of a transition that describes the modification of the variables is represented as a boolean relation, which suggests a succinct and efficient BDD representation.

In particular, variables appear explicitly and are not encoded in the stack symbols. The relation R with which a symbolic transition rule is labeled specifies how the rule transforms the variables. R is usually represented as a BDD.

It is straightforward to translate between symbolic PDSs and other simple specification methods, such as pseudo-code or control flow graphs. [Sch02] formalizes this, we just give an example.

Example 2.6. Consider the procedures in Figure 2.1. The procedure `main` calls the procedure `foo`.

```
procedure main

m0:  L := L · (L + 1)
m1:  call foo(L)
m2:  if G ≠ 0 then
         error
```

```
procedure foo(P)

f0:  if P even then
f1:      P := 0
     else
f2:      P := 561
f3:  G := P
```

Figure 2.1: Two simple procedures.

Procedure `foo` returns a value using the global variable G. Procedure `main` has a local variable L, procedure `foo` has a local variable P.

The transition rules of a corresponding symbolic PDS are shown in Figure 2.2. The set of initial configurations is given by stack symbol $m0$.

$$
\begin{array}{ll}
\langle m0\rangle \hookrightarrow \langle m1\rangle & (L' = L \cdot (L+1)) \\
\langle m1\rangle \hookrightarrow \langle f0, m2\rangle & (L'' = P' = L \ \wedge \ G' = G) \\
\langle m2\rangle \hookrightarrow \langle error\rangle & (G \neq 0) \\
\\
\langle f0\rangle \hookrightarrow \langle f1\rangle & (P \text{ even} \ \wedge \ G' = G) \\
\langle f1\rangle \hookrightarrow \langle f3\rangle & (P' = 0 \ \wedge \ G' = G) \\
\langle f0\rangle \hookrightarrow \langle f2\rangle & (P \text{ odd} \ \wedge \ G' = G) \\
\langle f2\rangle \hookrightarrow \langle f3\rangle & (P' = 561 \ \wedge \ G' = G) \\
\langle f3\rangle \hookrightarrow \langle\rangle & (G' = P)
\end{array}
$$

Figure 2.2: A simple symbolic PDS corresponding to Figure 2.1.

Figure 2.3 shows a corresponding control flow graph. Later, we focus on this graphical form of representation.

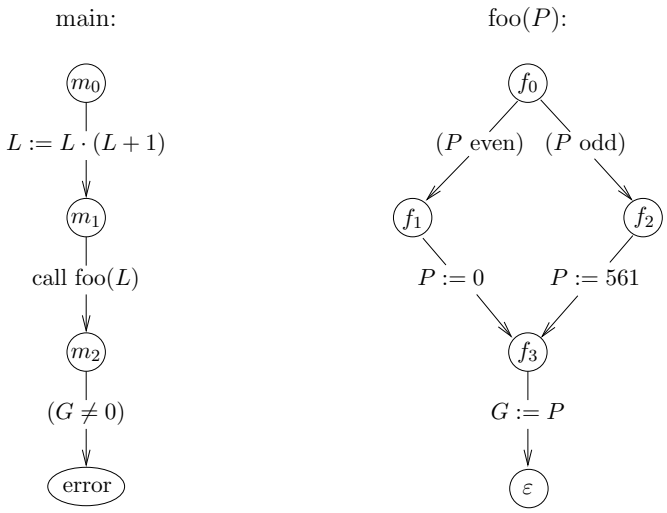

Figure 2.3: A symbolic PDS as a control flow graph.

We could mark the edges of this and similar figures with the corresponding pushdown rules. However, to keep notation intuitive, we rather write them in the form of program statements:

- Skips and assignments. As usual in programming languages, it is understood that variables not occurring on the left side of an assignment keep their values.
- Conditions. They are denoted in parentheses, e. g., (P even). Conditions usually correspond to the branch of an `if`-statement to be taken

if this condition holds. As the program runs, edges marked with a condition can only be taken if the condition holds at that time.

- Procedure calls, e. g., `call foo`(L). They correspond to pushdown rules with 2 control points on the right side.
- Procedure returns. They are denoted as edges whose target is ε, as depicted at the end of procedure `foo` in Figure 2.3.

Skips, assignment and conditions correspond to pushdown rules with one control point on the right side. Note that a `skip`-statement and a condition (1) have the same meaning, since both do nothing but keep the variable values.

2.3 Moped

Moped is a model checker for symbolic PDSs, developed in Stefan Schwoon's PhD thesis [Sch02]. It is based on the calculation of successor and predecessor configurations to a given pushdown configuration. The results are assembled in finite automata that accept the successor or predecessor configurations.

Moped accepts multiple input languages, including Java Bytecode [SSE05]. Programs with procedures, including recursion, are handled as well. Restrictions of Moped include the fact that finite range integers and booleans are the only currently implemented basic datatypes. After compiling the input, the system to be model-checked is a symbolic PDS as described in Section 2.1. The variable relations are represented as BDDs.

Moped has been proven to be very efficient and has been successfully used inside the SLAM project [Sch02].

2.4 Our work and related work

The goal of this thesis is to design and implement a CEGAR scheme for symbolic PDSs.

2.4.1 Moped in the SLAM project

Moped has already been successfully used in the SLAM project [Sch02, BR01]. However, it was only used as replacement of the model checker Bebop [BR00]. The exclusive task of Moped was to model-check the abstract boolean program and to deliver a counterexample if one exists. The

other steps in the CEGAR cycle, i. e., checking the counterexample for spuriousness, the generation of predicates [BR02] and constructing the abstract boolean program [BMMR01], were beyond Moped's responsibility.

2.4.2 Theorem provers vs. BDDs

In SLAM [BR01] as well as in related projects, theorem provers are used for checking counterexamples for spuriousness and for the generation of predicates. For instance, the projects in [BR02] and [HJMS02] call Simplify [DNS00]. The use of theorem provers can cause different problems:

1. Operation types might be restricted. For example, the proof system of [HJMM04] works only in cases where at most one multiplicand is a program variable.
2. The theorem provers easily become the bottleneck of the model checking process, as described in [HJMS02].
3. Gödel's incompleteness theorem excludes precise abstractions even in quite simple proof systems.

In contrast, we consider programs given as symbolic PDSs. The variable relations are encoded in BDDs. Therefore, the variable domains are finite, as it is the case in real-world computers.

As a consequence, we do not use theorem provers. Instead, we use BDDs throughout the CEGAR cycle. Thus, we can compute precise abstractions no matter what kind of expressions are contained in the programs to be model-checked. In principle, the finite-domain approach also enables us to discover overflow errors, i. e., errors that arise from the fact that real-world variables, too, comprise a limited number of bits [CKSY04].

A fundamental challenge of our approach is the fact that the generated predicates can only consist of expressions already present in the program to be model-checked. We use no a-priori knowledge about the meaning of the symbolic transition rules.

Finite variable domains have also been used in [CKSY04] and [CKSY05]. For the abstraction step as well as for the check of spuriousness, they build instances of SAT and solve them with a SAT solver. They do not use BDD techniques for the concrete program.

2.4.3 Craig interpolation

Craig interpolation has been used before in model checking applications. [McM05] gives an overview. Like [HJMM04], we use interpolants for the

predicate-generation step. In addition to the predicates themselves, this process also derives information about *where* the predicates are useful. Since we need to track predicates only at control points where they are relevant, the number of predicates per control point decreases. That makes the abstract system smaller and therefore faster to model-check [HJMM04].

In contrast to [HJMM04], we do not consider a special arithmetic proof system, but investigate pure propositional logics. We develop a theory of weakest and strongest interpolants that can be implemented by standard features of BDD libraries. This theory can be used directly for predicate generation, but also opens up a wide field of possible heuristics for finding meaningful predicates.

2.4.4 Multiple counterexamples

Another novelty of our work is the use of multiple counterexamples for predicate refinement. Whereas usually the model checker reports only one counterexample per CEGAR cycle, we can handle multiple counterexamples.

To this end, we access Moped's internal datastructures to obtain multiple counterexamples organized in a directed acyclic graph (DAG). Since our interpolation theory nicely generalizes to DAGs, our predicate-refinement step is able to rule out a whole DAG of spurious counterexamples.

Chapter 3

Craig Interpolation

Motivated by the use of Craig interpolants [Cra57] for abstraction in [HJMM04], we study Craig interpolants in this section. In contrast to [HJMM04], where a specialized arithmetic proof system is used, we investigate pure propositional logics.

The following definition is adapted from [McM03].

Definition 3.1. *Let (A, B) be a pair of formulas with $A \wedge B$ unsatisfiable. An interpolant for (A, B) is a formula J with the following properties:*

- $A \models J$,
- $J \wedge B$ *is unsatisfiable and*
- J *refers only to the common variables of A and B.*

The Craig Interpolation Theorem [Cra57] states that there is always an interpolant. It is not unique though. This raises the question if there exist *special* interpolants. In fact, it turns out that there is a weakest (and also a strongest) interpolant.

Definition 3.2. *The weakest interpolant for (A, B) is the interpolant for (A, B) that is implied by all interpolants for (A, B). It is denoted by $WI(A, B)$. The strongest interpolant for (A, B) is the interpolant for (A, B) that implies all interpolants for (A, B). It is denoted by $SI(A, B)$.*

The notion of "the weakest interpolant" and "the strongest interpolant" is well-defined. We show it for the weakest interpolant:

Let $\mathcal{J}$ be any set of interpolants for (A, B). Then $J' := \bigvee_{J_i \in \mathcal{J}} J_i$ is an interpolant as well. Notice that we deal only with finitely many variables, therefore J' can be equivalently written as some finite formula, namely the weakest interpolant.

It is clear by definition that $SI(A, B) \models WI(A, B)$.

In the Sections 3.1 and 3.2, we investigate properties of the weakest (respectively, strongest) interpolants. Section 3.3 formalizes interpolant computation schemes for formulas with a special structure. Those schemes will be used later in this thesis.

3.1 Weakest interpolants

Theorem 3.3. *Let C, G be formulas over a set of variables X. Let $Z \subseteq X$. Let $F = \forall Z(C \rightarrow G)$. Then F is the weakest formula over $X \setminus Z$ s.t. $F \wedge C \models G$.*

Proof.

1. The notion of "the weakest formula" is well-defined in our context, as can be seen by a similar argument as for the definition of the weakest interpolant.

2. Show: $F \wedge C \models G$. Since $F \models (C \rightarrow G)$, it suffices to show: $(C \rightarrow G) \wedge C \models G$. That follows from $(C \rightarrow G) \wedge C \equiv C \wedge G$.

3. Let $\tilde{F}$ be a formula over $X \setminus Z$ with $\tilde{F} \wedge C \models G$. Let a be any assignment to $X \setminus Z$ with $a(F) = 0$. Show: $a(\tilde{F}) = 0$. Since $a(F) = 0$, the assignment a can be extended to an assignment a^+ to X s.t. $a^+(C \wedge \neg G) = 1$. Since $\tilde{F} \wedge C \models G$, $a^+(\tilde{F}) = a(\tilde{F}) = 0$. □

As can be shown with the previous theorem, the weakest interpolant $WI(A, B)$ can be expressed quite simply:

Theorem 3.4. *Let (A, B) be a pair of formulas over X with $A \wedge B$ unsatisfiable. Let Z be the variables that occur in B, but not in A. Let $J = \forall Z.\neg B$. Then $J \equiv WI(A, B)$.*

Proof.

1. The formula J refers only to variables common to A and B, because the variables that only occur in B have been quantified out.

2. Show: $A \models J \models \neg B$. To see this, regard J as being formed according to Theorem 3.3: choose $F \equiv J$, $G \equiv \neg B$, $C \equiv 1$. Then, J is the weakest formula over $X \setminus Z$ that implies $\neg B$. Any interpolant I is a formula over $X \setminus Z$ that implies $\neg B$. Therefore $I \models J$. Because $A \models I$, we also have $A \models J$. Thus, we can conclude that J is an interpolant, and for any interpolant I we have $A \models I \models J \models \neg B$. Therefore, J is indeed the weakest interpolant $WI(A, B)$. □

It can be inferred from Theorem 3.4 that the weakest interpolant for (A, B) actually depends only on B and the variables occurring in A, not on A itself. We will now investigate some relations between the weakest interpolants for similar formula pairs. Those relations will become useful for efficiently computing weakest interpolants.

Theorem 3.5. *Let $A \wedge B \wedge C$ be unsatisfiable. Let Y be the variables that occur in B, but not in A. Then $WI(A, B \wedge C) \equiv \forall Y(B \rightarrow WI(A \wedge B, C))$.*

Proof. By Theorem 3.4, $WI(A \wedge B, C) \equiv \forall Z.\neg C$, where Z is the set of variables that occur only in C. By the same theorem, $WI(A, B \wedge C) \equiv \forall(Z \cup Y)(\neg B \vee \neg C)$.
Since the variables in Z do not occur in B, $WI(A, B \wedge C)$ can be rewritten as $\forall Y(\neg B \vee \forall Z.\neg C) \equiv \forall Y(B \rightarrow WI(A \wedge B, C))$. □

Corollary 3.6 (Tracking Property)**.** $WI(A, B \wedge C) \wedge B \models WI(A \wedge B, C)$.

Proof. $WI(A, B \wedge C) \wedge B \equiv \forall Y(B \rightarrow WI(A \wedge B, C)) \wedge B$
$\models (B \rightarrow WI(A \wedge B, C)) \wedge B \models WI(A \wedge B, C)$. □

Example 3.7. Consider three formulas A, B, C over X_1, X_2, X_3, X_4, where X_1, X_2, X_3, X_4 are pairwise disjoint. The formula A only contains variables in $X_1 \cup X_2$. There are similar restrictions to the variables in B and C. Those dependencies are depicted on the left side of Figure 3.1 by means of a graph. (For now, ignore the direction of the edges.)

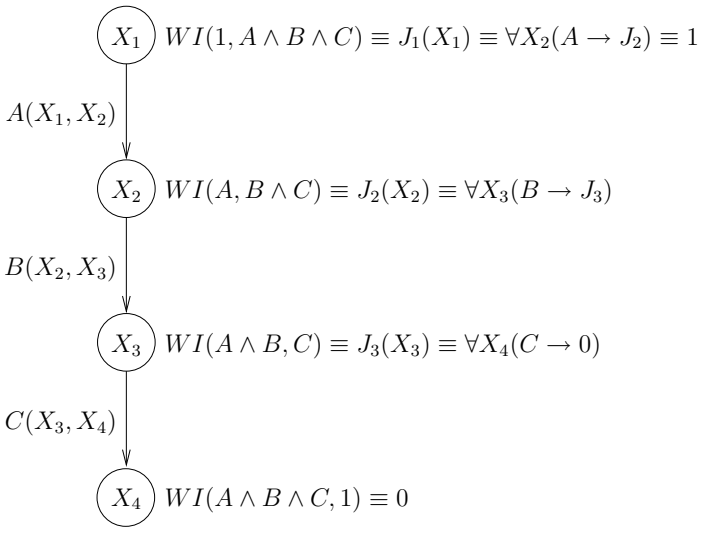

Figure 3.1: The weakest interpolants can be computed from bottom to top.

We assume $A \wedge B \wedge C$ to be unsatisfiable. Therefore, weakest interpolants are defined. The right side of Figure 3.1 shows relations between different

weakest interpolants. Those relations can be used to compute the weakest interpolants from bottom to top.

The tracking property (Corollary 3.6) yields $A \models J_2$, $J_2 \wedge B \models J_3$, $J_3 \models \neg C$.

Corresponding to Theorem 3.5, there is a theorem for disjunctions instead of conjunctions:

Theorem 3.8. *Let $A \wedge (B \vee C)$ be unsatisfiable. Then $WI(A, B \vee C) \equiv WI(A, B) \wedge WI(A, C)$.*

Proof. Let Z be the set of variables occurring in $B \vee C$, but not in A. Let X be the variables occurring in B, but not in A. Let Y be the variables occurring in C, but not in A. Then, $Z = X \cup Y$.

According to Theorem 3.4, $WI(A, B \vee C) \equiv \forall Z.\neg(B \vee C)$. This can be rewritten as $\forall Z(\neg B \wedge \neg C) \equiv \forall Z.\neg B \wedge \forall Z.\neg C \equiv \forall X.\neg B \wedge \forall Y.\neg C \equiv WI(A, B) \wedge WI(A, C)$. □

Example 3.9. Consider an unsatisfiable formula $F \equiv (A \wedge D \wedge F) \vee (A \wedge C \wedge E) \vee (B \wedge E)$. The variables occurring in A, B, C, D, E, F are, similarly as in Example 3.7, depicted by means of a graph (Figure 3.2).

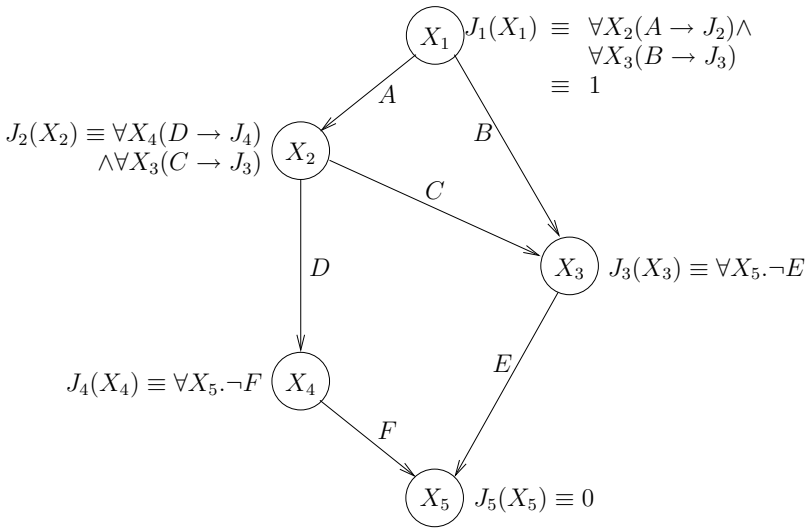

Figure 3.2: A formula depicted as a DAG. The weakest interpolants can be computed from bottom to top.

This graph is, in fact, a DAG. Notice that each path from top (X_1) to bottom (X_5) corresponds to a conjunction appearing in the disjunctive normal form of F. In fact, the DAG can be seen as a succinct representation of the formula F.

Consider now any node of the DAG (e. g. X_3) along with the DAG above (X_1, X_2, X_3) and the DAG below (X_3, X_5). If F_1 is the formula belonging to the DAG above $((A \wedge C) \vee B)$ and F_2 is the formula belonging to the DAG below (E), then $F_1 \wedge F_2$ is unsatisfiable.[1] Therefore it is meaningful to talk about interpolants (for (F_1, F_2)) at particular *cut points* (e. g. X_3) of the DAG.

The formulas J_i in Figure 3.2 are weakest interpolants in that sense. Using Theorems 3.5 and 3.8, they can be computed from bottom to top, similarly as in Figure 3.1.

3.2 Strongest interpolants

There are corresponding theorems for strongest (instead of weakest) interpolants. We state them without proof here, since the theorems and proofs are in many ways "dual" to the ones in the previous section.

Theorem 3.10 (dual to Theorem 3.3)**.** *Let C, F be formulas over a set of variables X. Let $Z \subseteq X$. Let $G = \exists Z(F \wedge \neg C)$. Then G is the strongest formula over $X \setminus Z$ s.t. $F \models C \vee G$.*

Theorem 3.11 (dual to Theorem 3.4)**.** *Let (A, B) be a pair of formulas over X with $A \wedge B$ unsatisfiable. Let Z be the variables that occur in A, but not in B. Let $I = \exists Z.A$. Then $I \equiv SI(A, B)$.*

Theorem 3.12 (dual to Theorem 3.5)**.** *Let $A \wedge B \wedge C$ be unsatisfiable. Let Y be the variables that occur in B, but not in C.*
Then $SI(A \wedge B, C) \equiv \exists Y(SI(A, B \wedge C) \wedge B)$.

Corollary 3.13 (Tracking Property)**.** $SI(A, B \wedge C) \wedge B \models SI(A \wedge B, C)$.

Theorem 3.14 (dual to Theorem 3.8)**.** *Let $(A \vee B) \wedge C$ be unsatisfiable. Then $SI(A \vee B, C) \equiv SI(A, C) \vee SI(B, C)$.*

Example 3.15. Consider the same formula as in Example 3.9. Figure 3.3 is dual to Figure 3.2. The I_i denote strongest interpolants and can be computed from top to bottom using Theorems 3.12 and 3.14.

3.3 Interpolant computation schemes

In this section, we formalize the interpolation schemes from the Examples 3.9 and 3.15, since we use them later on.

[1] This is because $F_1 \wedge F_2 \models F$.

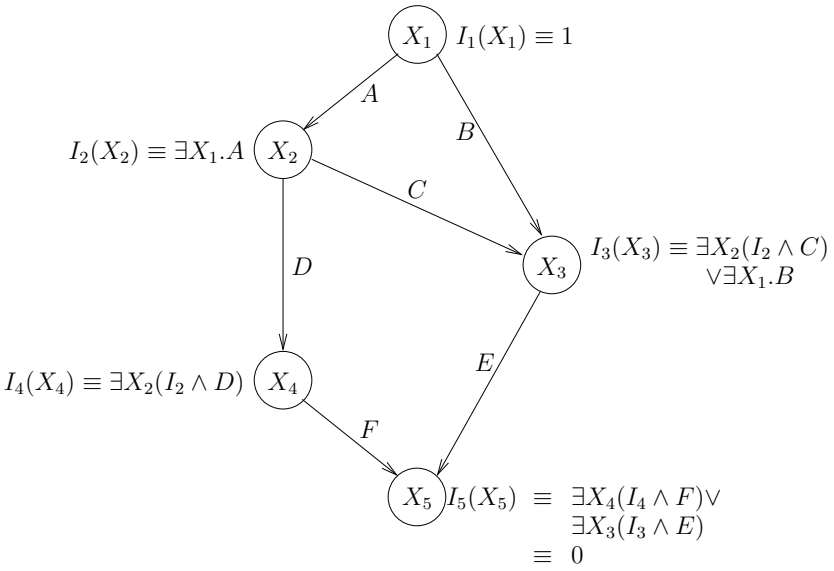

Figure 3.3: The strongest interpolants can be computed from top to bottom.

3.3.1 Polarized DAGs

Let $G = (V, E, (n_\top, n_\bot))$ be a *polarized DAG*. By that we mean:

- (V, E) is a finite directed acyclic graph.
- $n_\top, n_\bot \in V$.
- $\forall n \in V : \exists$ a path from $n_\top$ over n to $n_\bot$.

Figure 3.3 shows an example of a polarized DAG.

Notice that each node $n \in V$ induces a *super-DAG* and a *sub-DAG*. The super-DAG of n is the polarized DAG $G_n^\uparrow = (V_n^\uparrow, E_n^\uparrow, (n_\top, n))$ induced by the nodes from which n is reachable. The sub-DAG of n is the polarized DAG $G_n^\downarrow(V_n^\downarrow, E_n^\downarrow, (n, n_\bot))$ induced by the nodes which are reachable from n.

Let G be marked as follows:

- For all nodes $n \in V$: X_n is a set of propositional variables, such that $i \neq j \Longrightarrow X_i \cap X_j = \emptyset$.
- For all edges $e = (n_1, n_2) \in E$: $F_e(X_{n_1}, X_{n_2})$ is a formula over $X_{n_1} \cup X_{n_2}$.
- Let be $P = (e_1, \ldots, e_p)$ be a path from $n_\top$ to $n_\bot$. Let

$$F(P) := \bigwedge_{i=1}^{p} F_{e_i}.$$

Then $F(P)$ is unsatisfiable.

This marking defines a mapping from a polarized DAG G to a formula as follows:

$$F(G) := \bigvee_{P \text{ is a path from } n_\top \text{ to } n_\bot} F(P)$$

Because the $F(P)$ are all unsatisfiable, $F(G)$ is unsatisfiable as well.

Theorem 3.16. *Let $G = (V, E, (n_\top, n_\bot))$ be a polarized DAG as defined above. Let $G_n^\uparrow, G_n^\downarrow$ be super- and sub-DAGs of any node n.*
Then $F(G_n^\uparrow) \wedge F(G_n^\downarrow)$ is unsatisfiable.

Proof.

$$F(G_n^\uparrow) = \bigvee_{P_n^\uparrow \text{ is a path from } n_\top \text{ to } n} F(P_n^\uparrow)$$

$$F(G_n^\downarrow) = \bigvee_{P_n^\downarrow \text{ is a path from } n \text{ to } n_\bot} F(P_n^\downarrow)$$

By the associative law, it follows

$$\begin{aligned} F(G_n^\uparrow) \wedge F(G_n^\downarrow) &\equiv \bigvee_{\substack{P_n^\uparrow \text{ is a path from } n_\top \text{ to } n, \\ P_n^\downarrow \text{ is a path from } n \text{ to } n_\bot}} \big(F(P_n^\uparrow) \wedge F(P_n^\downarrow)\big) \\ &\equiv \bigvee_{P \text{ is a path from } n_\top \text{ over } n \text{ to } n_\bot} \big(F(P)\big) \\ &\models \bigvee_{P \text{ is a path from } n_\top \text{ to } n_\bot} \big(F(P)\big) \\ &\equiv F(G). \end{aligned}$$

$F(G)$ is unsatisfiable, as seen before. □

Because of Theorem 3.16 it is meaningful to talk about interpolants at a *cut point* n: The pair $\big(F(G_n^\uparrow), F(G_n^\downarrow)\big)$ is unsatisfiable. The following two subsections describe how to compute those interpolants.

3.3.2 Weakest interpolants

Using Theorems 3.5 and 3.8, weakest interpolants $J_n(X_n)$ at cut points n can be computed inductively (from bottom to top) as follows:

$$J_n(X_n) :\equiv 0 \qquad \text{(if } n = n_\bot\text{)}$$

$$J_n(X_n) :\equiv \bigwedge_{\substack{e = (n, n') \text{ is an} \\ \text{outgoing edge of } n}} \big(\forall X_{n'}(F_e(X_n, X_{n'}) \to J_{n'}(X_{n'}))\big) \qquad \text{(else)}$$

Example 3.9 can be seen as an illustration of this.

3.3.3 Strongest interpolants

Using Theorems 3.12 and 3.14, strongest interpolants $I_n(X_n)$ at cut points n can be computed inductively (from top to bottom) as follows:

$$I_n(X_n) :\equiv 1 \qquad \text{(if } n = n_\top\text{)}$$

$$I_n(X_n) :\equiv \bigvee_{\substack{e=(n',n) \text{ is an} \\ \text{ingoing edge of } n}} \big(\exists X_{n'}(I_{n'}(X_{n'}) \wedge F_e(X_{n'}, X_n))\big) \qquad \text{(else)}$$

Example 3.15 can be seen as an illustration of this.

Chapter 4

Our approach

In this section, we describe our design of the CEGAR loop (Figure 1.1) adapted to symbolic PDSs. The central idea is that we use BDDs throughout the CEGAR loop: For the concrete system, for the abstract system and for the predicates.[1]

We focus on monolithic programs in this section, i. e., programs without procedure calls. In Chapter 5 we generalize to full symbolic PDSs.

4.1 Predicate abstraction

Initially, we have no predicates. Therefore, our initial abstraction eliminates all data. Only the control flow, given as a PDS, is kept. Technically, this is done by replacing the BDDs of the rules by the BDD representing the boolean value 1 ($\equiv true$).

In later passes of the CEGAR loop, we do have predicates. As in [HJMM04], we use *localized* predicates, i. e., predicates along with information where they are useful. This information comes naturally as a by-product of the predicate-refinement step.

Technically, for each control point of the program, we maintain a list of predicates. Given a concrete statement along with its associated control points and predicate lists, we compute an abstract version of this statement by existential abstraction. This is illustrated in the following example.

Example 4.1. The left side of Figure 4.1 shows a simple statement ($Y := 2X$) along with its two control points $s1, s2$ and associated predicate lists.

To get the abstract version of the statement with respect to the given pred-

[1] Like Moped, we use Cudd [Som98] as BDD library.

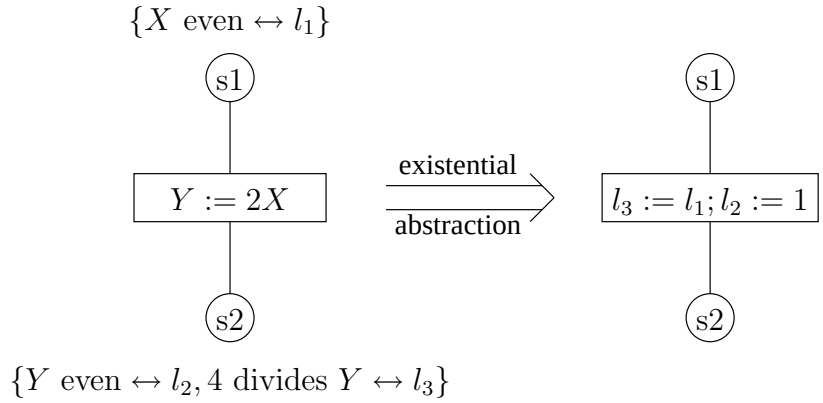

Figure 4.1: Existential Abstraction.

icates, we form the conjunction of all terms and quantify out (existentially) the concrete variables X, Y:

$$\begin{aligned} absRel &\equiv \exists X, Y(\quad (X \text{ even } \leftrightarrow l_1) \wedge (Y = 2X) \wedge \\ &\qquad\qquad\quad (Y \text{ even } \leftrightarrow l_2) \wedge (4 \text{ divides } Y \leftrightarrow l_3)) \\ &\equiv (l_1 \leftrightarrow l_3) \wedge l_2 \end{aligned}$$

This relation over the abstract variables can be formulated in a more "programming language-like" fashion as depicted on the right side of Figure 4.1.

The abstract statement sets the variable l_2 to 1. That is due to the fact that the predicate corresponding to l_2 (Y even) will hold after execution of the concrete statement. This is because the value of Y is the result of a multiplication of an integer with the number 2.

As for the other predicates, Y will be divisible by 4 if and only if X is even. Therefore, the abstract variable l_3 is assigned the value of l_2.

In terms of symbolic PDSs, the concrete transition rule looks as follows:

$$\langle s1 \rangle \hookrightarrow \langle s2 \rangle \quad (Y' = 2X \ \wedge \ X' = X)$$

The abstract version of this rule is then

$$\langle s1 \rangle \hookrightarrow \langle s2 \rangle \quad (l_3' = l_1 \wedge \ l_2' = 1).$$

In general, existential abstraction leads to an overapproximation: If a trace is defined as the sequence of control points visited during a program execution, then all traces of the concrete program (and maybe more) are also traces of the abstract program. [CGJ+00] has a more formal proof of this fact and includes examples of spurious traces.

We start now with a running example in order to illustrate some points of our abstraction refinement scheme.

Example 4.2. Consider the code in Figure 4.2. We want to model-check the fact that `error` is not reachable.

```
1:  X := X · (X + 1)
2:  if a = 0 then
3:       Y := 2
    else
4:       while Y mod 2 ≠ 0 do
5:            Y := Y + 1
6:  X := X + Y
7:  if X mod 2 ≠ 0 then
         error
```

Figure 4.2: Example code.

For now, assume that we track the predicate $a \neq 0$ at all locations and that this predicate corresponds to the abstract boolean variable l.[2] Existential abstraction produces abstract code like in Figure 4.3. Figure 4.4 shows a corresponding control flow graph.

```
1:  skip
2:  if l = 0 then
3:       skip
    else
4:       while ? do
5:            skip
6:  skip
7:  if ? then
         error
```

Figure 4.3: Abstracted example code.

4.2 Model checking the abstract system

After an abstract system has been built in the predicate-abstraction step (see Section 4.1), Moped is used to model-check it. Currently, the only allowed specification is a reachability check.[3] Typically, the code to be model-checked has been instrumented by the user such that an error label can only be reached if the program violates the specification.

[2] Tracking $a \neq 0$ everywhere is not typical for our CEGAR scheme, but it keeps the example simple.

[3] The most recent redesigned implementation of Moped currently allows only reachability checks.

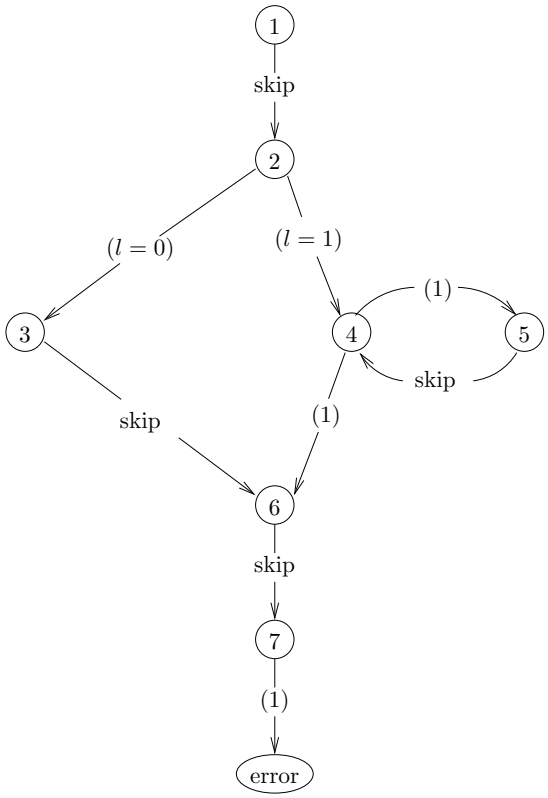

Figure 4.4: Control flow graph for the abstract program.

To model-check the specification, Moped performs a search through the state space of the program. It records the reachable states (i. e., reachable control points along with possible variable values). In order to be able to eventually report a counterexample, Moped also records information about *how* program states can be reached. To this end, the reachable states are collected in *witness graphs*. The directed edges of those graphs link the reachable states and are marked with the rule used to transform the old state into the new state. In [RSJMar], the details of those witness graphs are elaborated.

If Moped finds that the (error) label in the specification is not reachable in the abstract program, we conclude (see Section 1.5) that it is not reachable in the concrete system either. This result is reported to the user.

If, on the other hand, Moped finds that the (error) label is reachable in the abstract program, Moped is able to deliver a counterexample, i. e., a trace through the abstract program reaching the (error) label. In order to find such a trace, Moped uses the previously built witness graph: Starting from the error label, it follows the edges of the witness graph in opposite direction until a start state has been found.

Very often, the abstract system contains multiple traces leading to the (error) label. Furthermore, Moped's witness graphs might already contain several of them. If the witness graph contains a real (non-spurious) counterexample, it should be picked from the witness graph as described above. This abstract trace should be translated into a corresponding real trace and should be reported to the user.[4]

If, on the other hand, the witness graph contains only spurious counterexamples, one could also take a single counterexample and use it for further predicate refinement. This is the traditional approach in CEGAR (see Figure 1.1). We have found, however, that the predicate-refinement step (see Section 4.3) can benefit from multiple (spurious) counterexamples.

Moped's witness graphs naturally organize multiple counterexamples in a directed acyclic graph (DAG): If we take the parts of the witness graph that witness the reachability of the error label, we essentially arrive at a DAG of counterexamples. It has a unique top node (the control point where the program starts) and a unique bottom node (the error label).[5] Each path through this DAG is an abstract error trace. Thus, constructing a DAG of counterexamples comes with no additional cost: Moped already builds it in the model-checking phase, when the state space of the program is searched.

Example 4.3. Continuing Example 4.2, we assume that we ask Moped to model-check the (abstract) program in Figure 4.4. Moped then produces a witness graph that can also be understood as a DAG of counterexamples. Moped might produce the DAG shown in Figure 4.5. This DAG contains three different traces leading to the error label. Since Moped computes reachable states, it also computes the possible variable values for each control point. Those predicates are denoted in brackets, e. g., $[l = 1]$. Internally, they are represented as BDDs.

4.3 Spurious counterexamples and predicate refinement

In Section 1.5 we sketched what to do with an abstract counterexample: First, check whether the counterexample is real or spurious. Then, if it is spurious, it is used for further refinement of the abstraction.

We deviate from this scheme in the following respects:

- We have a DAG of counterexamples instead of a single counterexample.
- The categorization step and the refinement step are closely integrated. We use the interpolation theory from Chapter 3 for both tasks.

[4] Currently, for time reasons, we only report an abstract trace in the non-spurious case.
[5] In Section 3.3 we called this a *polarized DAG*.

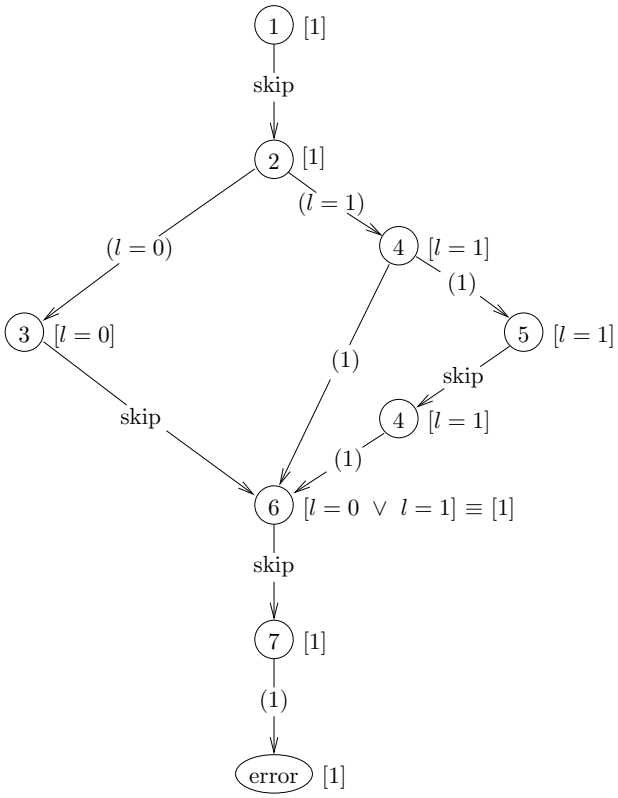

Figure 4.5: DAG of counterexamples produced by Moped.

4.3.1 A DAG of counterexamples as a formula

From the model-checking step, we have a DAG of counterexamples. We keep the structure of this DAG, but eliminate the abstract variable values and replace the abstract rules by their corresponding concrete rules. We get an object similar to the one depicted in Figure 4.6.

Note the similarity between Figure 4.6 and Figure 3.2 on page 18. They are both polarized DAGs. We will further tighten this connection and establish a correspondence between the spuriousness of a DAG of counterexamples and the unsatisfiability of a formula. Finally, we will show that if the counterexamples in the DAG are all spurious, then we are able to rule out the whole DAG of spurious counterexamples by choosing interpolants as additional predicates.

The program variables can have different values at each control point. We already distinguish between *new* and *old* variable values in the transition rules of a symbolic PDS. In a generalization of this idea, we can think of distinct variable sets for each control point.

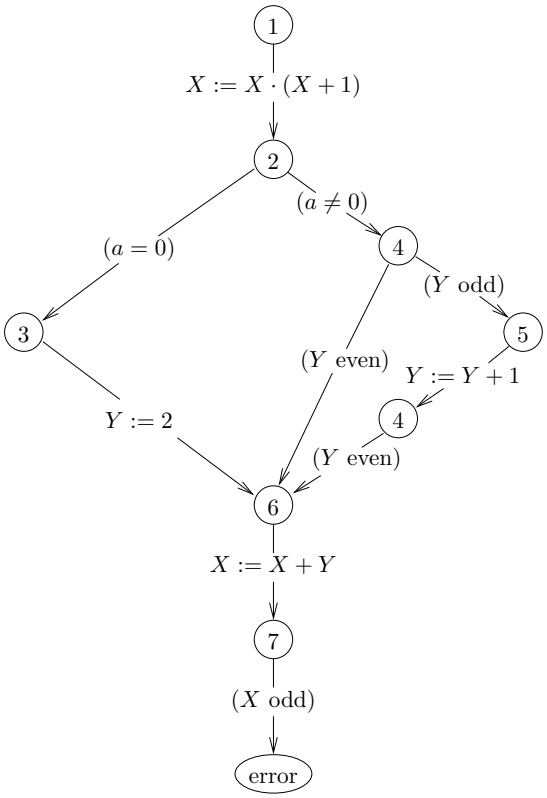

Figure 4.6: DAG of counterexamples.

A path through the DAG (a trace) can then be seen as a conjunction of the formulas on the edges. This trace is spurious if and only if the corresponding conjunction is unsatisfiable.

In the example of Figure 4.6, the formula to the trace 1–2–4–6–7–error is the conjunction

$$\begin{array}{lll} & (a_2 = a_1) & \wedge\ (X_2 = X_1 \cdot (X_1 + 1)) \quad \wedge\ (Y_2 = Y_1) \\ \wedge & (a_4 = a_2 \neq 0) & \wedge\ (X_4 = X_2) \quad \wedge\ (Y_4 = Y_2) \\ \wedge & (a_6 = a_4) & \wedge\ (X_6 = X_4) \quad \wedge\ (Y_6 = Y_4 \text{ even}) \\ \wedge & (a_7 = a_6) & \wedge\ (X_7 = X_6 + Y_6) \quad \wedge\ (Y_7 = Y_6) \\ \wedge & (a_{error} = a_7) & \wedge\ (X_{error} = X_7 \text{ odd}) \quad \wedge\ (Y_{error} = Y_7) \end{array}$$

A DAG can contain multiple traces. We say, *a DAG is spurious* if it only contains spurious traces. In other words, it is spurious if the corresponding conjunctions are all unsatisfiable. Therefore it is natural to consider the disjunction of the conjunctions: This formula is unique to each DAG (up to equivalence) and is unsatisfiable if and only if the DAG is spurious.

The DAG of Figure 4.6 contains 3 different traces. Hence, the formula to that DAG is the disjunction of 3 trace formulas.

Note that we arrived at the same correspondence between a DAG and a formula as in Section 3.3 and Example 3.9. We conclude our observations in the following proposition:

Proposition 4.4. *A DAG of counterexamples is spurious if and only if its corresponding formula is unsatisfiable.*

4.3.2 Weakest interpolants

We can compute weakest interpolants in a bottom-up manner as in Section 3.3.2: Start from the bottom of the DAG (the error label) and assign it the boolean value 0. Then, in each step, follow the edges in opposite direction and use quantification over the target variables to obtain the interpolant over the source variables.[6] Finally, in the spurious case, this method will assign the boolean value 1 to the top of the DAG.

In the non-spurious case, the computation of interpolants must fail. This is discovered at the top of the DAG: Only if the weakest interpolant computation assigns 1 to some node, the DAG beneath is spurious by itself. So, if the top node is not assigned 1, the DAG contains a real counterexample. Thus, computation of the interpolants gives us a criterion of spuriousness.

Proposition 4.5. *A DAG of counterexamples is spurious if and only if the bottom-up weakest interpolant computation assigns* 1 *to the top node.*

Figure 4.7 continues the previous example by displaying the weakest interpolants in brackets (e. g., $[X \text{ even}]$). Since the counterexamples in the DAG are all spurious, the top node is assigned the interpolant 1.

We want to characterize the weakest interpolants in a more intuitive manner. Consider, e. g., the nodes 3 and 6 and the edge connecting them. The weakest interpolant at node 3 $[X \text{ even}]$ is the weakest formula that, assuming the execution of $Y := 2$, implies the interpolant at node 6 $[X + Y \text{ even}]$. Thus, the weakest interpolant is some form of weakest precondition.

If a node has multiple outgoing edges, as does node 2, then a similar computation can be performed for each edge. Considering all outgoing edges, the weakest interpolant is then the conjunction of those separately computed formulas (cf. Theorem 3.8). For node 2, the weakest interpolant is the conjunction of $[X \text{ even} \vee a \neq 0]$ and $[X \text{ even} \vee a = 0]$. This amounts to $[X \text{ even}]$. The formula computed this way is the weakest formula that, assuming execution of *any* outgoing rule, implies the interpolant of the respective target.

Those relations between the weakest interpolants yield – when combined – the following proposition:

[6] Quantification over some variables is a BDD library function.

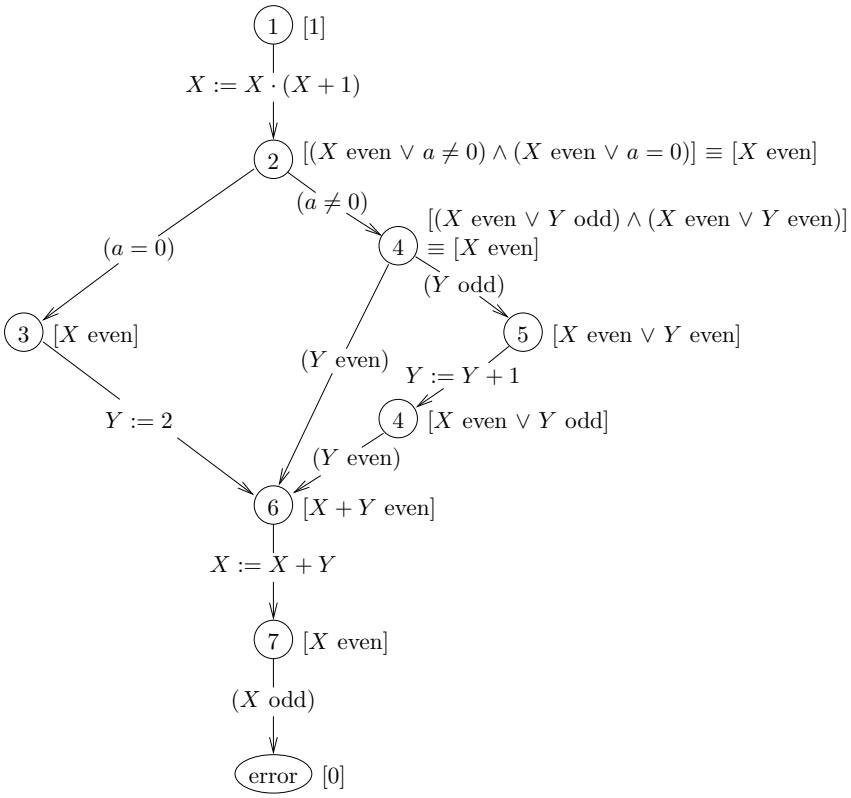

Figure 4.7: Weakest interpolants.

Proposition 4.6. *The weakest interpolant at some node n in the DAG is the weakest predicate over the program variables that excludes all traces in the DAG between n and the bottom node.*

In the example of Figure 4.7 that means for node 2: The error label is not reachable by any path through the DAG if and only if the program variables satisfy the predicate $[X \text{ even}]$.

Those interpolants exclude – if chosen as new localized predicates in our abstraction – the whole DAG of counterexamples. To see this, consider any edge in the DAG along with the two nodes connected by it, e. g., nodes 3 and 6 in Figure 4.7. As seen before and also by Corollary 3.6, the weakest interpolant at the source $[X \text{ even}]$ implies, together with the rule on the edge $(Y := 2)$, the weakest interpolant at the target $[X + Y \text{ even}]$. Therefore, in an existential abstraction as described in Section 4.1, this relation will be kept: Whenever the abstract variable representing the source interpolant is 1, the abstract variable representing the target interpolant must also be 1, if the corresponding abstract rule is executed.

Since the weakest interpolant at the top node is 1 and the weakest inter-

polant at the bottom node is 0, it is not possible for the abstract program to reach the error label along any trace in the DAG. If the error label is still reachable, it must be along some other trace which is not contained in the DAG.

Proposition 4.7. *If all weakest interpolants are added to the abstraction as new localized predicates, then the whole DAG of spurious counterexamples is excluded.*

As a conclusion, the weakest interpolants exhibit two nice properties:

- They give a criterion for the spuriousness of counterexamples.
- They are able to rule out the DAG of spurious counterexamples in subsequent abstractions.

4.3.3 Strongest interpolants

As in Chapter 3, there is a duality between weakest and strongest interpolants. One important relationship between strongest and weakest interpolants is clear by definition: The strongest interpolants imply their corresponding weakest interpolants.

Dual to the bottom-up weakest interpolants computation scheme used in the previous section, there is the top-down strongest interpolants computation scheme described in Section 3.3.3.

We now take the propositions from Section 4.3.2 and rephrase them in terms of strongest interpolants instead of weakest interpolants.

Proposition 4.8. *A DAG of counterexamples is spurious if and only if the top-down strongest interpolant computation assigns* 0 *to the bottom node.*

Proposition 4.9. *The strongest interpolant at some node n in the DAG is the strongest predicate over the program variables that is implied by all traces in the DAG between the top node and n.*

Proposition 4.10. *If all strongest interpolants are added to the abstraction as new localized predicates, then the whole DAG of spurious counterexamples is excluded.*

The last proposition states that both weakest and strongest interpolants are able to exclude the spurious counterexamples at hand, i. e., in the DAG. So, there is no a-priori preference which kind of interpolants to choose. Notice that no harm is done if both weakest and strongest interpolants are added to future abstractions, except for a larger number of predicates to be tracked.

4.3.4 Other interpolants

Experiments show (see Chapter 6) that neither weakest nor strongest interpolants are the best choice for all applications. Rather, strongest and weakest interpolants form a frame inside which possibly different interpolants can be heuristically chosen. In this section we set out this frame and state the properties the interpolants should fulfill:

1. They should be valid interpolants, i. e., if I and J respectively denote the strongest and weakest interpolant, then any interpolant K must fulfill $I \models K \models J$. In particular, this implies that the interpolant at the top of the counterexample DAG must be 1 and the interpolant at the bottom must be 0.

2. Let n_1, n_2 be two nodes in the DAG, and let them be connected by a (concrete) rule $C(X_1, X_2)$. Let $K_1(X_1)$ and $K_2(X_2)$ be interpolants for n_1 and n_2 respectively. They should fulfill $K_1 \wedge C \models K_2$.

Both weakest and strongest interpolants fulfill those requirements. In particular, the tracking properties (Corollaries 3.6 and 3.13) guarantee the second requirement. The second requirement is necessary for actually excluding the DAG, as explained in Section 4.3.2.

However, the tracking property does not hold automatically for all combinations of strongest, weakest or other interpolants. The following example illustrates this fact.

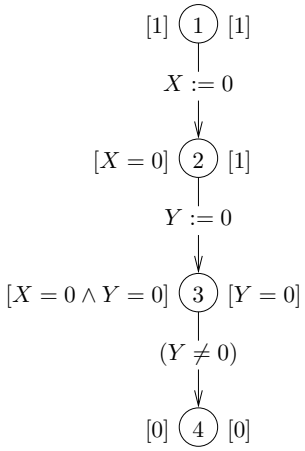

Figure 4.8: The tracking property does not hold for all interpolants.

Example 4.11. Consider the trace in Figure 4.8. It is clearly spurious. Strongest and weakest interpolants, respectively, are displayed in brackets to the left and to the right of the control points.

Looking at the edge connecting nodes 2 and 3, we see that the tracking property holds for the strongest interpolants:

$$[X = 0] \quad \wedge \quad Y := 0 \qquad \models \qquad [X = 0 \wedge Y = 0]$$

Similarly, it holds for the weakest interpolants:

$$[1] \quad \wedge \quad Y := 0 \qquad \models \qquad [Y = 0]$$

However, it does not hold if the weakest interpolant is chosen for node 2 and the strongest for node 3:

$$[1] \quad \wedge \quad Y := 0 \qquad \not\models \qquad [X = 0 \wedge Y = 0]$$

Thus, the weakest interpolant for node 2 and the strongest for node 3 are not suitable predicates for ruling out this spurious trace.

Knowing this, we propose the following algorithm for computing suitable interpolants K_i in a bottom-up fashion.

1. Compute strongest interpolants I_i for all nodes n_i.
2. Set $K_\perp \equiv 0$ for the bottom node $n_\perp$.
3. Let n_1, n_2 be nodes in the DAG and $K_2(X_2)$ an already computed interpolant for n_2. Let $C(X_1, X_2)$ the rule connecting n_1 and n_2.

   Set $K_1(X_1)$ heuristically, but in a way that guarantees the following properties:

   (a) $I_1 \models K_1$
   (b) $K_1 \wedge C \models K_2$
4. Repeat step 3 in a bottom-up fashion until $K_\top$ is set to 1 for the top node $n_\top$.

There exists also a corresponding "top-down" algorithm.

From this point of view, weakest and strongest interpolants are just two possible heuristics. Other heuristics are presented in Chapter 6.

Chapter 5

Predicates in the context of procedures

In Chapter 4 we discussed our approach for an implementation of the CEGAR loop. There, we assumed a monolithic program, i. e., a program without explicit procedures.

The presence of procedures complicates that. In this section, we discuss how to generalize the concepts of Chapter 4 to the procedural case.

We proceed as follows. In Section 5.1 we introduce some problems due to the fact that procedures bring along their own scope. In Section 5.2 we describe predicate-generation schemes for procedural counterexamples. In Section 5.3 we precisely define how to compute an abstract PDS given the concrete PDS and the previously computed predicates. Section 5.4 mentions our implementation of this CEGAR scheme for symbolic PDSs.

5.1 Modular counterexample DAGs

Example 5.1. Consider again the example program of Section 2.1, given in Figure 2.3 on page 11. It features a procedure `foo` that is called by another procedure `main`. Since this particular program is non-recursive, we could inline the procedure call and use Moped to model-check an abstract version of the program. In analogy to Figure 4.6 on page 29, Moped could give us a DAG of spurious counterexamples as shown in Figure 5.1.

Inlining causes several problems though:

- the *recursion problem*: A main feature of PDSs is the fact that recursion is allowed. Disallowing it would severely impair the usefulness of the PDS model.

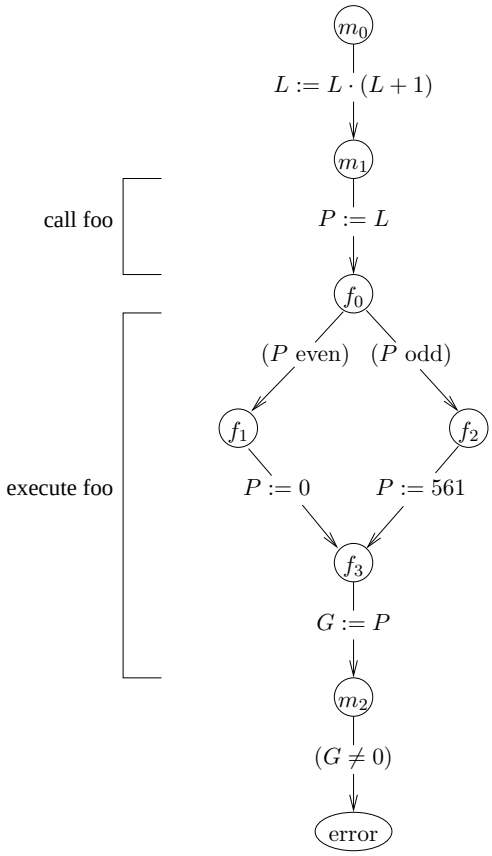

Figure 5.1: DAG of counterexamples when the procedure call is inlined.

- the *blow-up problem*: Even in the non-recursive case, a PDS might blow up if procedures are inlined.
- the *scope problem*: We could go ahead as follows: For the abstract program, we could allow recursion and procedure calls. Then, Moped reports a (finite) counterexample DAG with explicit sub-DAGs for called procedures (see, e. g., Figure 5.2). We could neglect this procedure information and afterwards inline the procedures "conceptually" to get a single monolithic counterexample DAG. The problem is that our standard method for computing interpolants might produce predicates over variables that are out of scope.

For an illustration of the scope problem, consider the strongest interpolant at cut point f_1 in Figure 5.1. It can be written as

$$I_{f_1}(P, L) \equiv \exists L_0(P = L = L_0 \cdot (L_0 + 1)).$$

The problem is that I_{f_1} depends on L, which is out of the scope of procedure `foo`.

For a modular interprocedural analysis, we rather wish to obtain predicates that are independent of the particular calling context. The solution has been repeatedly described in the literature, e. g. in [HJMM04]: We need to compute the effect of a procedure in terms of the arguments passed to it, i. e., we relate the "output" of a procedure to its "input".

In our framework, the input of a procedure comprises the initial values of the called procedure in a push statement. By "initial values", we mean the initial values of the local variables, along with the global variable values after the call. The output of a procedure consists of the global variable values after the pop statement.

A predicate over the input and the output can thus describe the effects of a procedure. We will also need predicates in the called procedure itself. Here, instead of the output, we consider the current local and global variable values, i. e., we relate the input to the current procedure values.

The starting point of our counterexample analysis is a *modular* counterexample DAG, which includes the called procedures as explicit sub-DAGs. Such a DAG can be obtained using Moped's internal witness graphs (see Section 4.2).

Figure 5.2 (analog to Figure 4.6) illustrates this with the previous example. The node marked with $f_0 m_2$ acts as *pivot* control point. This is the place where we will obtain predicates describing relations between variables of the calling procedure and variables of the called procedure.[1]

5.2 Predicate generation

In the model-checking step, Moped delivers us a (modular) counterexample DAG as described in the previous section. This DAG has to be analyzed. More precisely, it has to be checked if the DAG is spurious. If so, then suitable predicates have to be computed that exclude the DAG in subsequent abstractions.

As in the non-procedural case, we tightly integrate the spuriousness check and the predicate generation. This is the subject of this section. Again, as in Section 4.3, there are two basic predicate-generation schemes:

- top-down strongest interpolants computation
- bottom-up weakest interpolants computation

Those schemes are described in the Sections 5.2.1 and 5.2.2. There, we assume a counterexample, where one procedure calls another, and then re-

[1] Strictly spoken such predicates must be out of scope. To allow for parameter passing, we need such predicates at defined points though.

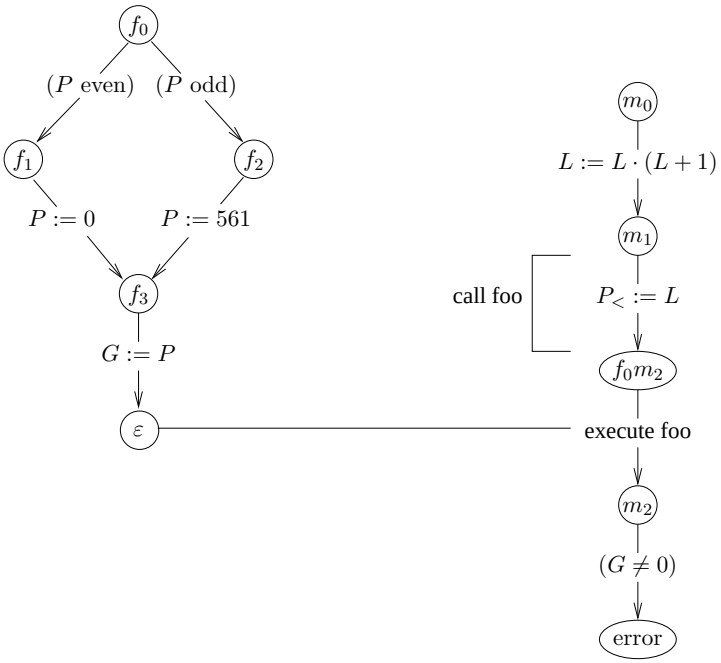

Figure 5.2: A modular counterexample DAG.

sumes execution after the called procedure finishes. In Sections 5.2.3 and 5.2.4, we generalize this to other counterexample structures. Section 5.2.5 discusses how to combine different structures.

5.2.1 Strongest interpolants

As in the non-procedural case, we base our top-down interpolants computation scheme on the method of Section 3.3.3. The difference lies, of course, in the called procedures. As motivated in Section 5.1, we wish to keep a relation between the current values and the input values of a procedure. For this purpose, we do not start with the predicate $[true]$ at the entry point of the called procedure, but with a predicate describing equality between the input values and the current values of the procedure. This predicate must hold, since the current procedure values at the entry point *are* the input values.

The reason for introducing extra variables for the input values is that we can keep the previously described computation scheme. The input values will never be quantified out, therefore, at each point in the procedure, the relation to the input values will be kept. Figure 5.3 illustrates this by continuing Example 5.1.

In this figure, the variable $P_<$ denotes the input value of the local variable P.

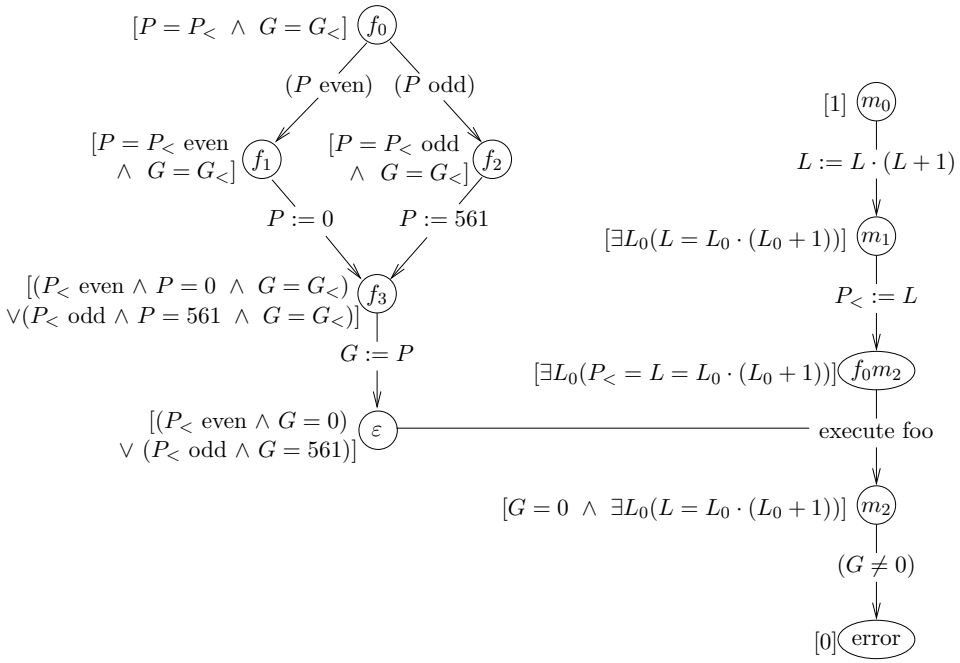

Figure 5.3: Strongest interpolants for a modular DAG.

Similarly, $G_<$ denotes the input value of the global variable G. The usual top-down scheme is employed for computing most of the predicates. For node m_2, we have a deviation: The only edge leading to m_2 is not marked with a transition rule but with "execute foo". However, since the predicate at node ε describes the input-output relation of the procedure (its effect), we can take this relation and regard it as an ordinary statement. Figure 5.4 illustrates this.

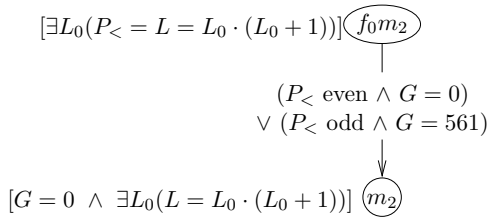

Figure 5.4: A procedure run as a statement.

For the description of the general case, we assume that the modular counterexample DAG contains a procedure `m` that calls a procedure `f` and continues execution after `f` has finished. We assume for notational convenience that our symbolic PDS has a single global variable G. The procedure `m` has a single local variable L, and `f` has a single local variable P. Figure 5.5 shows how to compute strongest interpolants I_n for a modular DAG.

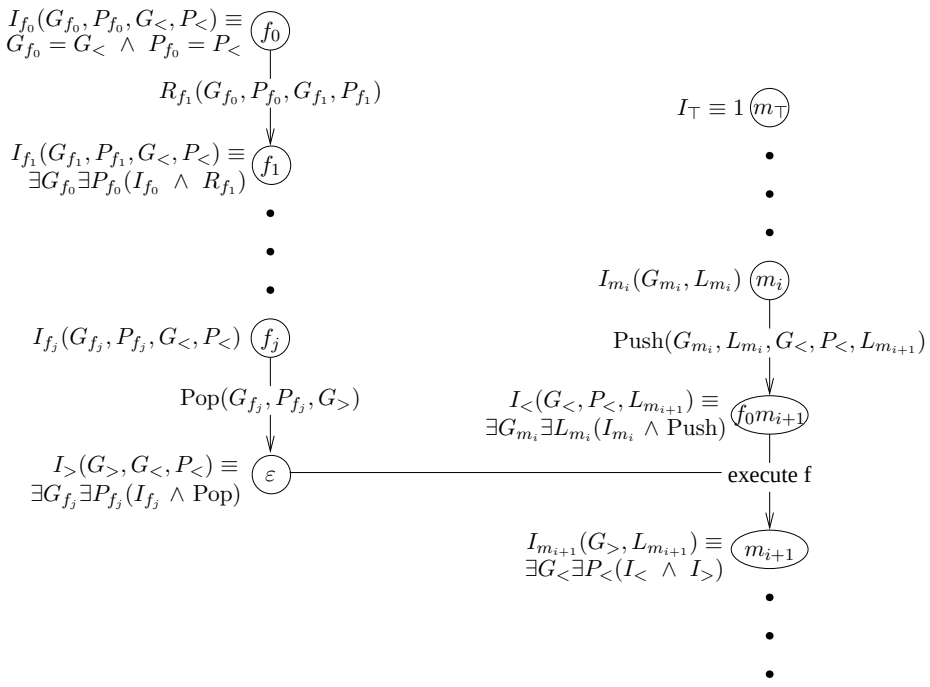

Figure 5.5: Strongest interpolants for a modular DAG.

This top-down computation scheme first computes the strongest interpolant I_{m_i} at point m_i. Then, the strongest interpolant $I_<$ at the pivot point $f_0 m_i$ is computed using the push rule of this PDS. This interpolant $I_<$ describes the relation between the input values of `f` ($G_<$ and $P_<$) and the local variable L of `m`.

The strongest interpolants in `f` are computed similarly. However, the relation to the input values $G_<$ and $P_<$ is kept during the computation. At the beginning, this relation is just an equality. The normal pushdown rule R_{f_1} (one control point on the right side) at the beginning of `f` serves as an example of how to continue the computation downwards. At the end of `f`, the node ε is reached. Its interpolant $I_>$ describes an input-output relation of `f`, i. e., its effect in this counterexample.

This effect of `f`, in turn, can be used for computing the interpolant $I_{m_{i+1}}$ in `m` after returning from `f`.

The spuriousness of a modular counterexample DAG will turn out at the bottom: Similarly as before, it is spurious if and only if the bottom node is assigned 0. Otherwise, there is a real counterexample and the computed predicates are, in fact, not interpolants.

In the spurious case, it is crucial to argue why the calculated predicates rule out the given counterexample DAG. As in the non-procedural case (Sec-

tion 4.3), the idea is to have *true* predicates at the top and a *false* predicate at the bottom. An interpolant, together with the rule of an outgoing edge, implies the interpolant at the target node. In Chapter 4 we called this the *tracking property*. By looking at the way we compute the I_n, it is easy to verify that it still holds in the procedural case.

In particular, note that the predicate at the pivot point and the procedure effect together imply the predicate at the node after the return from the procedure:

$$I_< \ \wedge \ I_> \quad \models \quad I_{m_{i+1}}$$

Since existential abstraction is used, those implications carry over to the abstract program: Whenever the abstract variable representing the source interpolant is 1, the abstract variable representing the target interpolant must also be 1, if the corresponding abstract rule is executed. Hence, a counterexample in the current DAG will not occur again in future counterexamples.

In Section 5.3 we write down in detail how the abstract program is computed in the procedural case.

5.2.2 Weakest interpolants

We want to generalize the bottom-up weakest interpolant computation scheme from Section 3.3.2 to the procedural case. Such a scheme should be related to the scheme from the previous section in the following ways:

- The predicates J_n from the bottom-up scheme should have the same *signature* as the predicates I_n from the top-down scheme. By signature, we mean the set of variables on which a predicate depends.
- As in the non-procedural case, we want $I_n \models J_n$ to hold for each control point n.
- Similar to the previous section, we want to guarantee the tracking property, in particular: $J_< \wedge J_> \models J_{m_{i+1}}$.

The handling of the cases involving normal pushdown rules and pop rules is quite straightforward. However, the cases involving the pivot point need some additional attention. Figure 5.6 shows our bottom-up scheme in the procedural case.

First, the weakest interpolant $J_{m_{i+1}}$ is computed. For $J_<$, we have the following requirements:

(1) Its signature should be $J_<(G_<, P_<, L_{m_{i+1}})$.

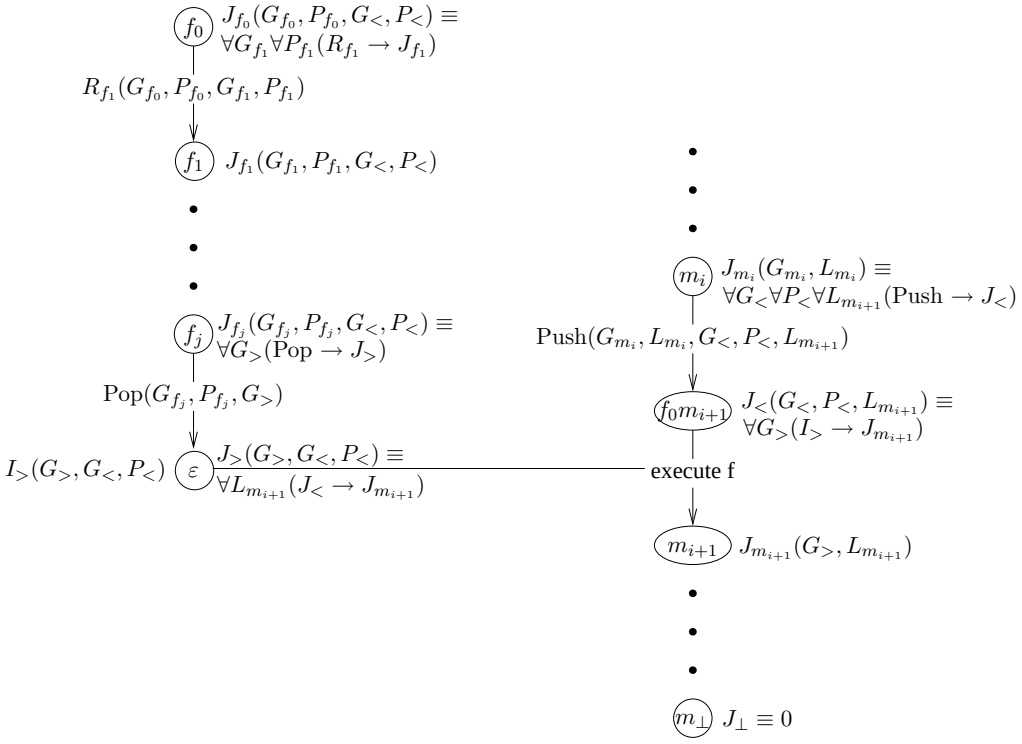

Figure 5.6: Weakest interpolants for a modular DAG.

(2) $I_< \models J_<$.

(3) $J_< \wedge J_> \models J_{m_{i+1}}$, where $I_> \models J_>$.

For now, assume $J_> \equiv I_>$. Then, according to Theorem 3.3, the weakest formula $J_<$ fulfilling requirements (1) and (3) is

$$J_< \equiv \forall G_>(I_> \to J_{m_{i+1}}).$$

Note that $J_<$ could only get stronger if $I_>$ is replaced by a weaker formula. Therefore, the above $J_<$ is indeed the weakest formula fulfilling requirements (1) and (3), even without the assumption $J_> \equiv I_>$. Since $I_<$ also fulfills (1) and (3), we have $I_< \models J_<$, i. e., requirement (2).

Once $J_<$ is fixed, the predicate $J_>$ has to be computed. Again, we want $J_>$ to be as weak as possible and to fulfill the following requirements:

(1) Its signature should be $J_>(G_>, G_<, P_<)$.

(2) $I_> \models J_>$.

(3) $J_< \wedge J_> \models J_{m_{i+1}}$.

According to Theorem 3.3, the weakest formula $J_>$ fulfilling requirements (1) and (3) is

$$J_> \equiv \forall L_{m_{i+1}}(J_< \rightarrow J_{m_{i+1}}).$$

Since $I_>$ also fulfills (1) and (3), we have $I_> \models J_>$, i. e., requirement (2).

On a more intuitive level, we use the most precise (strongest) description of the procedure effect ($I_>$) to jump from $J_{m_{i+1}}$ to $J_<$. Thus we get the weakest $J_<$ possible. Once $J_<$ is fixed, we can weaken $I_>$ to $J_>$ by overapproximating the procedure effect in terms of the given counterexample DAG: We need not say more about $J_>$ than that it transforms $J_<$ to $J_{m_{i+1}}$.

Figure 5.7 shows this bottom-up weakest interpolants computation scheme applied to the previous example program.

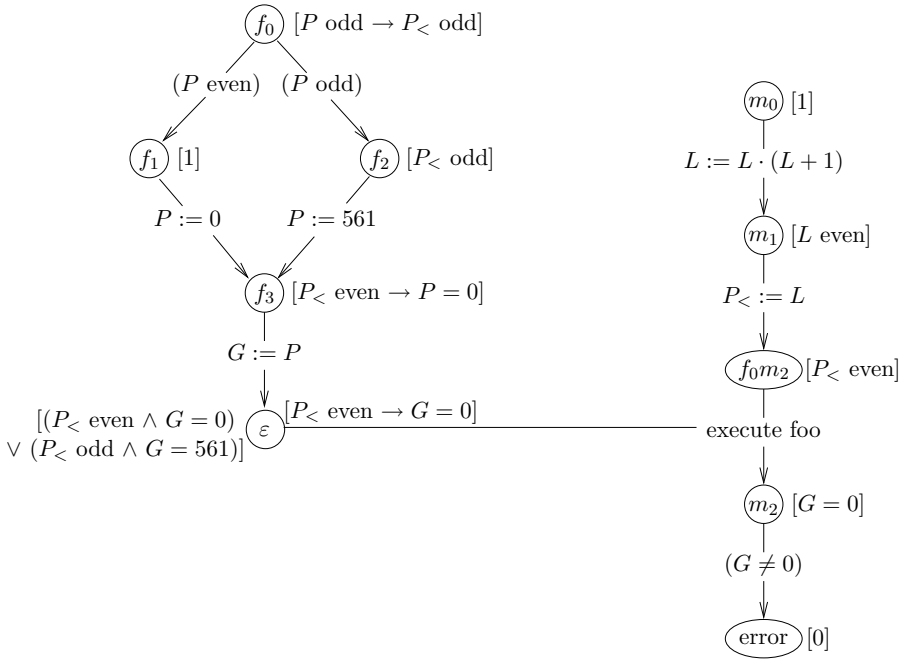

Figure 5.7: Weakest interpolants for a modular DAG.

The spuriousness of the DAG will turn out at the top of the main procedure: It is spurious if and only if the top node is assigned 1. Otherwise, there is a real counterexample and the computed predicates are, in fact, not interpolants.

The predicates at the entry point of the called procedure will always hold. This is because, by construction, they are implied by their top-down counterparts, which describe an equality of the input variables and the current variables. As mentioned before, this equality must hold, since at the entry point, the input values *are* the current values.

As in the previous section, the tracking property also holds by construction of the predicates, i. e., in the spurious case, the current counterexample DAG is ruled out.

5.2.3 Nested counterexamples

In Sections 5.2.1 and 5.2.2, we assumed that, in the reported abstract counterexample, a main procedure `m` calls another procedure `f` and resumes execution after `f` has finished. We did not take into account that `f` might again call a procedure. In this section, we describe how to deal with such cases. We essentially argue that only small modifications of the derived predicate-generation schemes are necessary.

When reviewing the predicates that we derive in our schemes, one might suspect that the number of involved variables grows with the calling depth. For example, consider Figure 5.6: While, at node m_i, we have a predicate with the signature $J_{m_i}(G_{m_i}, L_{m_i})$, the predicate at node f_j has the signature $J_{f_j}(G_{f_j}, P_{f_j}, G_<, P_<)$. This is because in procedure `f`, we need to track the relation of the current variables to the input values.

Fortunately, in the case of another nested procedure call, the number of variables does not increase any further. To see this, assume that the procedure `f` calls another procedure `g` with a local variable Q. In this procedure, we will track a relation between the current variables G and Q to the input values $G_\ll$ and $Q_\gg$. In particular, the input values of `f`, i. e., $G_<$ and $P_<$, are *not* passed to `g`. Therefore, the signature of the predicates in `g` is $J(G, Q, G_\ll, Q_\gg)$.

That means, from the point of view of the procedure `g`, the variables $G_<$ and $P_>$ are *local* variables of `f`. In the abstraction step (see Section 5.3), such variables can be treated in the same way as the local variables of the outer scope.

5.2.4 Calling and not returning from a procedure

In the previous parts of Section 5.2, we assumed that an inner procedure finishes in the given counterexample. Afterwards, the outer procedure that once called the inner procedure continues execution. However, this might not always be the case. A counterexample might easily describe a trace where a procedures is called and an error label is reached in the inner procedure.

In this "push-only" case, it is not meaningful to talk about the input-output relation of the inner procedure, since it did not terminate and thus has no output values.

It turns out that ruling out such counterexamples does not constitute a

severe problem. To see this, consider again the example from Section 5.1 (Figure 5.1), where we argued that inlining does not generally serve our purposes: The predicates describing the global state of a symbolic PDS might involve local variables that are out of the scope of the currently active procedure.

However, this reasoning does not fully apply in cases where the program does not return to the scope of those local variables. In those cases, we are actually not interested in the values of local variables that are never looked at again.

Example 5.2. Consider the following very simple program:

```
procedure main                    procedure foo(P)

m0:  L := 0                       f0:  if P = 1 then
m1:  call foo(L)                            error
```

In a spurious counterexample, the main procedure calls `foo`, but is not resumed, since `foo` does not finish. Therefore, the procedure `foo` can be "conceptually" inlined in this counterexample. Figure 5.8 shows this spurious counterexample along with strongest (on the left side) and weakest (on the right side) interpolants.

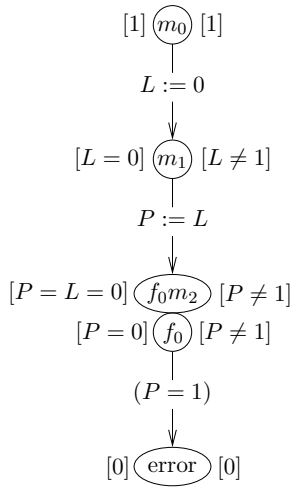

Figure 5.8: A spurious "push-only" counterexample with interpolants.

The idea is to quantify out local variables that are both irrelevant and out of scope in the counterexample. In the top-down strongest interpolants computation, we therefore quantify the local variable L out when moving from node f_0m_2 to node f_0. In the bottom-up computation, when moving from f_0 to f_0m_2, only a copy of the predicate is needed.

Figure 5.9 shows the general "push-only" case. We use the same notation as in Sections 5.2.1 and 5.2.2, i. e., a procedure `f` calls a procedure `m`, where `m` has a local variable L, `f` has a local variable P and both procedures share a global variable G. The notation $[V_1 \backslash V_2]$ denotes a renaming: V_1 is replaced by V_2. A generalization to arbitrarily many variables is straightforward.

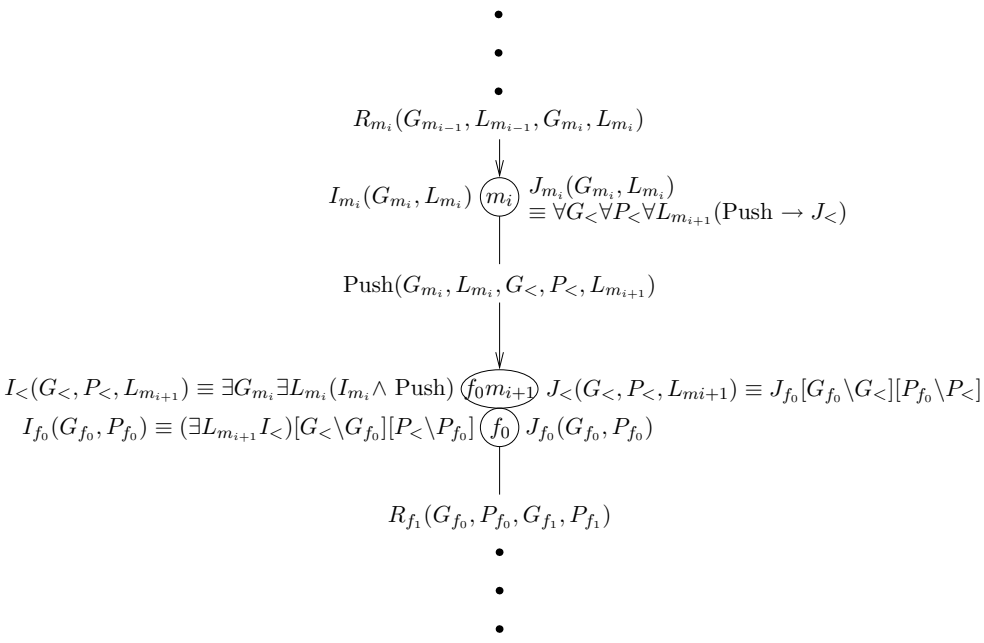

Figure 5.9: Predicate generation in a "push-only" counterexample.

The called procedure is conceptually inlined. In the top-down computation (depicted on the left side), the local variable L of the outer scope is quantified out at node f_0. This is done existentially, that means, we are only interested in the *possible* current variable values, regardless of how they relate to the local variables of the outer scope. In the bottom-up computation (depicted on the right side), the problem does not occur. As in the example, a (renamed) copy suffices.

Exactly as in the previously considered cases, spuriousness turns out the top resp. bottom of the DAG, if a bottom-up resp. top-down computation is performed.

To see, why the current counterexample is excluded in a subsequent abstraction (see Section 5.3), notice again the tracking property, here:

$$I_{m_i} \wedge \ \text{Push} \ \wedge (G_{f_0} = G_<) \wedge (P_{f_0} = P_<) \models I_{f_0}$$

This carries over to the abstract program, i. e., if the abstract variable corresponding to I_{m_i} is 1 before the push, then the abstract variable corresponding to I_{f_0} must be 1 after the push. This holds analogously for the

J_n.

In contrast to Sections 5.2.1 and 5.2.2, the predicates I_{f_0} and J_{f_0} do *not* necessarily hold when `f` is called.

5.2.5 Combination of different counterexample structures

In this Section 5.2, we discussed different counterexample structures that Moped could report us. In particular, we described how to perform predicate generation on those structures.

Of course, those structures could be mixed. The following example illustrates that.

Example 5.3. Consider a counterexample structure as follows:

- Procedure `m` calls procedure `f`.
- Procedure `f` calls itself recursively.
- The most recently called instance of `f` returns.
- The first instance of `f` resumes and reaches an error label.

In this case, we build a structure as in Figure 5.10 (page 48). Notice that we could have multiple counterexample traces (DAGs) instead of the vertical dots.

In general, if a procedure is called and it finishes, then this procedure is joined with the calling procedure as in Figure 5.2. If a procedure is called, but it does not finish, then it is joined with the calling procedure as in Figure 5.8.

5.3 Computing the abstract program

In this section, we write down in detail how to compute an abstract symbolic PDS, given the concrete symbolic PDS (including procedures) and predicates for each control point.

In Section 4.1 we described this process for programs without procedures. We emphasized that we use *existential abstraction*. In this section, we formally describe the general case including procedures.

We assume that we have, for each control point, predicates as computed in the previous sections. We reuse the notation of Figures 5.5 and 5.9. In particular, that means we have the same concrete variables (G, L, P) as before and one predicate I_n for each control point n. Therefore, we also

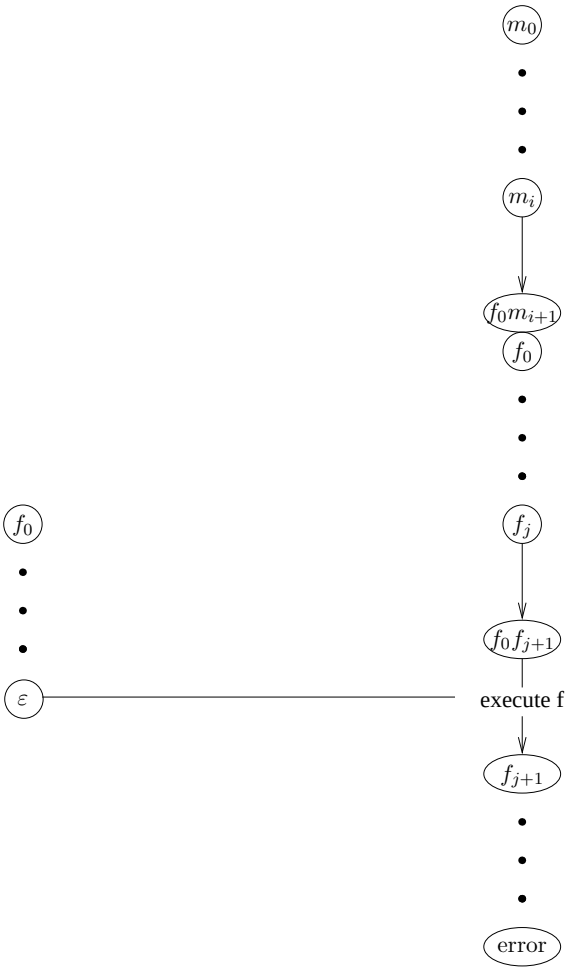

Figure 5.10: Combined structures.

have one abstract variable l_n in procedure `m` (resp. p_n in procedure `f`) for each control point. It will be clear how to generalize the situation to more predicates and variables.

Furthermore, we actually make the "pivot point" a real control point in the abstract program.

5.3.1 Concretizations

When computing an abstraction, one needs some link between the abstract variables and their meaning, i. e., their corresponding predicate over the concrete variables. We use the following notation here: For each node n, we compute a *concretization* denoted by $\{n\}$. It is a formula that links the abstract variables at that control point (in small letters) with their meaning

(in capital letters).

More precisely, we compute concretizations as follows.

$$\begin{array}{rcrcl}
\{m_i\} & \equiv & l_{m_i} & \leftrightarrow & I_{m_i}(G_{m_i}, L_{m_i}) \\
\{f_0 m_{i+1}\} & \equiv & l_{f_0 m_{i+1}} & \leftrightarrow & I_<(G_<, P_<, L_{m_{i+1}}) \\
\{f_0\} & \equiv & p_{f_0} & \leftrightarrow & I_{f_0}(G_{f_0}, P_{f_0}, G_<, P_<) \\
\{f_1\} & \equiv & p_{f_1} & \leftrightarrow & I_{f_1}(G_{f_1}, P_{f_1}, G_<, P_<) \\
\{f_j\} & \equiv & p_{f_j} & \leftrightarrow & I_{f_j}(G_{f_j}, P_{f_j}, G_<, P_<) \\
\{\varepsilon\} & \equiv & g & \leftrightarrow & I_>(G_>, G_<, P_<) \\
\{m_{i+1}\} & \equiv & l_{m_{i+1}} & \leftrightarrow & I_{m_{i+1}}(G_>, L_{m_{i+1}})
\end{array}$$

The l_n and the p_n are local abstract variables in `m` and `f`, respectively. The variable g is a global abstract variable.

In the case of more than one predicate at some location, those concretizations are combined with "$\wedge$". For instance, with two predicates I_{m_i} and J_{m_i} at m_i, one would have

$$\{m_i\} \equiv (l^I_{m_i} \leftrightarrow I_{m_i}(G_{m_i}, L_{m_i})) \wedge (l^J_{m_i} \leftrightarrow J_{m_i}(G_{m_i}, L_{m_i})).$$

In Section 7.2 we remark how to avoid repeated computations in this case.

5.3.2 Normal pushdown rules

The concrete rule

$$\langle f_0 \rangle \hookrightarrow \langle f_1 \rangle \quad (R_{f_1}(G_{f_0}, P_{f_0}, G_{f_1}, P_{f_1}))$$

is replaced by the abstract rule

$$\langle f_0 \rangle \hookrightarrow \langle f_1 \rangle \quad (r_{f_1}(p_{f_0}, p_{f_1})),$$

where

$$r_{f_1} \equiv \exists G_{f_0}, P_{f_0}, G_{f_1}, P_{f_1}, G_<, P_< \quad (\{f_0\} \wedge R_{f_1} \wedge \{f_1\}).$$

5.3.3 Pop rules

The concrete rule

$$\langle f_j \rangle \hookrightarrow \langle\rangle \quad (\mathrm{Pop}(G_{f_j}, P_{f_j}, G_>))$$

is replaced by the abstract rule

$$\langle f_j \rangle \hookrightarrow \langle\rangle \quad (\mathrm{pop}(p_{f_j}, g)),$$

where

$$\mathrm{pop} \equiv \exists G_{f_j}, P_{f_j}, G_>, G_<, P_< \quad (\{f_j\} \wedge \mathrm{Pop} \wedge \{\varepsilon\}).$$

5.3.4 Push rules

The concrete rule

$$\langle m_i \rangle \hookrightarrow \langle f_0, m_{i+1} \rangle \quad (\text{Push}(G_{m_i}, L_{m_i}, G_<, P_<, L_{m_{i+1}}))$$

is replaced by two abstract rules:

$$\langle m_i \rangle \hookrightarrow \langle f_0, (f_0 m_{i+1}) \rangle \quad (\text{push}(l_{m_i}, p_{f_0}, l_{f_0 m_{i+1}}))$$

and

$$\langle (f_0 m_{i+1}) \rangle \hookrightarrow \langle m_{i+1} \rangle \quad (\text{eval}(l_{f_0 m_{i+1}}, g, l_{m_{i+1}})),$$

where

$$\begin{aligned} \text{push} \equiv & \exists G_{m_i}, L_{m_i}, G_{f_0}, P_{f_0}, G_<, P_<, L_{m_{i+1}} \\ & (\{m_i\} \wedge \text{Push} \wedge (G_{f_0} = G_<) \wedge (P_{f_0} = P_<) \wedge \{f_0\} \wedge \{f_0 m_{i+1}\}) \end{aligned}$$

and

$$\text{eval} \equiv \exists G_<, P_<, L_{m_{i+1}}, G_> \quad (\{f_0 m_{i+1}\} \wedge \{\varepsilon\} \wedge \{m_{i+1}\}).$$

That means, in the case of a push rule, we need an additional *evaluation step* in order to include the procedure effect in the evaluation of the predicate at control point m_{i+1}.

Whereas the abstract variables can usually be local, a global variable g is needed to report the effect of the called procedure `f` back to the calling procedure.

5.4 Our implementation of the general scheme

We implemented the concepts of Chapter 5, which enables us to apply CEGAR to an arbitrary (potentially recursive) symbolic PDS.

We give examples in the next section when discussing several heuristics concerning the predicate generation. Some details about an efficient implementation are given in Chapter 7.

Chapter 6

Heuristics for interpolants

In Section 4.3.4 we listed the properties that the selected interpolants must fulfill in order to exclude the spurious counterexample DAG in subsequent abstractions.

Weakest and strongest interpolants fulfill those properties. Also other interpolants "in between" might fulfill them. However, compliance with those properties is not all we are interested in. Rather, we would like the selected interpolants to be powerful enough to exclude the counterexamples at hand *and also other counterexamples*.

In other words, we need good heuristics to decide automatically which interpolants have the potential of excluding many counterexamples. Informally spoken, such predicates are meaningful not only for the current counterexamples, but also for the program itself that is to be model-checked. Such predicates could be invariants or other predicates that the programmer had once in mind.

In this section, we present several heuristics that we came up with when experimenting with our implementation. Weakest and strongest interpolants form a frame for our search.

Section 6.1 presents the tracking property as a "guide" for finding good heuristics. In Section 6.2 we apply weakest and strongest interpolants as our simplest heuristics. The results are mixed, which motivates our search for alternatives. Section 6.3 suggests a heuristic that naturally conciliates between strongest and weakest interpolants. Section 6.4 presents a heuristic that can exploit the procedural structure of the program. Section 6.5 combines different heuristics and provides more examples illustrating the potential of the developed ideas. In Section 6.6 we apply our method to a Java program in order to document the gain of abstraction when dealing with real programs.

6.1 The tracking property as a guide

Assume that we have two control points n_1, n_2 and a concrete rule $C(X_1, X_2)$ describing how to modify the program variables to get from n_1 to n_2. Let $K_1(X_1), K_2(X_2)$ be interpolants for n_1, n_2, respectively, that were added to the abstraction in some (possibly different) cycles of the CEGAR loop.

If K_1 and K_2 were added in the same cycle of the CEGAR loop, then we probably required the tracking property

$$K_1 \wedge C \models K_2, \tag{6.1}$$

because the edge (n_1, n_2) probably occurred in the DAG of spurious counterexamples. However, we argue why implication (6.1) is desirable even if K_1, K_2 stem from different CEGAR cycles.

To see this, recall the meaning of an interpolant: An interpolant was added at some control point, because it excludes a DAG of spurious counterexamples below this point. In other words, if an interpolant is set to 1 at some control point of the abstract program, then the counterexamples in the DAG below are no longer feasible. Therefore, it is generally desirable if an interpolant predicate can be set to 1: Some counterexamples are excluded.

In addition, it is also good if the value 1 in the abstract program is *conveyed*, since the interpolants at the top are always 1. Because we construct the abstract program with existential abstraction, a 1 is conveyed from n_1 to n_2 if the tracking property (6.1) holds.

Therefore, we should keep the tracking property in mind when we design heuristics for choosing interpolants: We strive to choose interpolants which convey the value 1 in the abstract program.

As the next section shows, there is no simple and generally satisfying solution for this problem.

6.2 Weakest vs. strongest interpolants

Both weakest and strongest interpolants exclude the current counterexamples. We compare and test those simple heuristics in this section.

6.2.1 Theoretical considerations

Before implementing, we thought of two aspects speaking in favor of weakest interpolants.

- The weakest interpolants are often "more related" to the property to be checked. When considering reachability properties (as in this

thesis), this is mainly because the condition on which reachability of the error label depends is at the bottom of the DAG. So, the bottom-up weakest interpolant computation propagates this condition up the whole DAG. This is the case in Figure 4.7.

On the other side, the top-down strongest interpolant computation tends to lead to unintuitive predicates. Consider Figure 4.6. For instance, the strongest interpolant for node 2 would be $[Y = X \cdot (X + 1)]$, which seems to be much less related to the property to be checked than [Y even].

- The term *weak* can also be described as "easy to fulfill". This seems desirable, because as soon as an interpolant predicate is set to 1 in the abstract program, the error label is no longer reachable via the DAG that is ruled out by the interpolant.

  For instance, consider node 6 in Figure 4.7. Node 6 might also be reachable via other parts of the program which are not shown. At node 6, all that is needed to rule out the trace 6–7–error is the predicate $[X + Y$ even$]$. Choosing the weakest interpolant here "improves the odds" for other program parts to imply this predicate as well.

  In other words, with weakest interpolants one would expect that other spurious counterexamples with the same suffix are ruled out as well.

This reasoning, particularly the latter point, did not turn out to be generally relevant in practice. The tracking property provides a partial explanation. As stated in Section 6.1, we would like the tracking property

$$K_1 \wedge C \models K_2$$

to hold.

Above, we paraphrased "weak" as "easy to fulfill". In terms of the tracking property, we would argue that with a weak K_2 the likelihood for the tracking property to hold is increased. However, there is a dual argument saying that a strong K_1 would help it hold. In conclusion, the tracking property does not provide a definite answer to the question if weakest or strongest interpolants are better.

6.2.2 Experiments

We have done various experiments with weakest and strongest interpolants. Here, we cite two typical set-ups.

Example 6.1 (Bubble-Sort)**.** Figure 6.1 shows pseudo-code for a bubble-sort algorithm.

```
for i = 1 to n − 1
     for j = 1 to n − i
          if a[j] > a[j + 1] then
               swap(a[j], a[j + 1])
if ∃1 ≤ i < n : a[i] > a[i + 1] then
     error
```

Figure 6.1: A bubble-sort algorithm.

The array a is filled with nondeterministic values at the beginning. In other words, the array a is the input. We want to model-check the fact that `error` is not reachable.

For this experiment, the array a has 4 elements ($n = 4$), and each array element has 5 bits. We added interpolants according to three different strategies:

- weakest interpolants only
- strongest interpolants only
- both weakest and strongest interpolants

We compared those strategies with respect to three quantities:

- the number of CEGAR cycles needed to exclude all spurious counterexamples
- the maximum number of predicates at some control point
- the time used for the CEGAR loop

Figure 6.2 shows the results.

	weakest interp.	strongest interp.	both interp.
# of CEGAR cycles	55	13	8
max. # of predicates	113	12	34
time	15 sec	0.15 sec	0.8 sec

Figure 6.2: Bubble-sort results.

As can be seen, the weakest-interpolant-only heuristic is clearly inferior to both other strategies. This result can be interpreted as indication that this bubble-sort program is easier to analyze in a top-down fashion than in a bottom-up fashion.

As for the strongest-interpolant-only heuristic, we remark that at most 12 predicates are tracked at the same time, i. e., the abstract program needs 12

bits (see Section 7.1). The original concrete program, on the other side, has $5 \cdot 4 = 20$ bits.

If both weakest and strongest interpolants are added, then only 8 CEGAR cycles are needed. This can be explained by the tracking property (see Section 6.1). Many abstract 1s are conveyed, i. e., the tracking property

$$K_1 \wedge C \models K_2,$$

often holds. This is because K_1 could be a strong predicate (a strongest interpolant) and K_2 could be a weak predicate (a weakest interpolant). On the other hand, the number of predicates is higher, because we track both weakest and strongest interpolants.

No matter which heuristic is applied, the number of CEGAR cycles and the maximal number of predicates do not increase with the size of the array elements.[1] This is because the program contains only ">"-comparisons. Thus, the predicates are also based on those comparisons and are not sensitive to the element size.

This bubble-sort example gives some limited insights into the character of weakest and strongest interpolants. Our finding that a top-down analysis is faster and more appropriate in this example, reminds us of an experimental result of [Sch02], stating that, when applying Moped to the unabstracted code, a forward search (called $post^*$) usually yields better results than a backward search (called pre^*). Also interesting to note is the fact that weakest and strongest interpolants can be combined without severe performance problems.

However, the results in this bubble-sort example are not satisfying, because abstraction does not pay off. Applying Moped to the unabstracted code yields the best results in this example.

We explain that as follows. The correctness of this bubble-sort algorithm crucially depends on the complete execution of the loop. Possible invariants of the loops are hard to find without any a-priori knowledge about the structure of the algorithm. In addition, there are no major subtasks that are irrelevant for the property. We conclude that this example seems not suitable for abstraction in general.

Example 6.2 (Loop Invariant)**.** Figure 6.3 shows an example for a loop that tracks a simple loop invariant.

The variable g is "nondeterministically fixed" at the beginning, in other words, it is the input. We want to model-check the fact that `error` is not reachable.

[1] Runtime increases though.

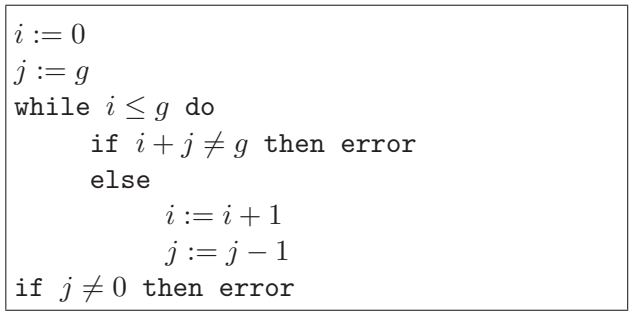

Figure 6.3: Code with a loop invariant.

We applied again the three heuristics from the previous example. In this case, the weakest-interpolant-only heuristic is clearly superior to the strongest-interpolant-only heuristic.

More precisely, the weakest-interpolants-only heuristic needs only 2 refinement cycles to prove that `error` is not reachable. This is done by only 1 predicate per control point. When the both-interpolants heuristic is used, then, again, 2 refinement cycles suffice, while at most 3 predicates per control point are needed. In both cases, the CEGAR time is low and negligible compared to the set-up time used for translating the code into a pushdown system with BDDs.

The strongest-interpolants-only heuristic, on the other hand, fails to detect the relation between the loop invariant ($i + j = g$) and the property to be checked after the loop ($j = 0$). Even though the CEGAR loop terminates successfully, this takes considerably more time. The number of cycles, the number of predicates and the runtime crucially depend on the number of bits per variable. In our experiments, we found that if n is the number of bits per variable, then the number of refinement cycles is roughly 2^{n+1} and the number of predicates per control point equals 2^n. Runtime increases similarly. We measured 8 seconds with $n = 6$.

With weakest interpolants, abstraction does pay off in this example. Even though the example is small enough that Moped can model-check it very quickly even without abstraction, there is more work to be done without abstraction. Moped essentially simulates the loop 2^n times in its search, where n is the number of bits per variable.

Abstraction pays off even more if we add a complicated procedure in front of the shown code (not shown). Moped must analyze this complicated procedure, whereas the weakest interpolant bottom-up computation does not consider it, since it reaches the interpolant 1 already between the complicated procedure and the shown code.

In conclusion of this example, the necessary predicates for proving unreach-

ability of `error` are already in the code. Thus, this example should be suitable for abstraction. The weakest-interpolants heuristic indeed finds the proof without actually fully executing the loop.

A conclusion of this section is that the gain of abstraction can extremely vary. We do not expect a result of the form "Abstraction improves performance by x %".

6.3 Conciliated interpolants

In the previous section we gave evidence that neither the weakest nor the strongest interpolants are the "best" interpolants in the sense that they are more useful in CEGAR than any other interpolants. Rather, we have argued before that we should strive to find interpolants that are meaningful not only with respect to the counterexamples at hand, but for the program itself.

In this section, we describe a heuristic that naturally conciliates between strongest and weakest interpolants. The basic idea is again Craig interpolation: We interpolate between the strongest and the weakest interpolants computed before. As a desired side effect, the predicates get simpler and hopefully closer to the invariants that the programmer had in mind.

The following theorem describes how to interpolate between two formulas.

Theorem 6.3. *Let $S(X,Y)$ be a formula over $X \cup Y$, let $W(Y,Z)$ be a formula over $Y \cup Z$. Let $S \models W$. Let $I(Y) \equiv \exists X.S$ and $J(Y) \equiv \forall Z.W$. Then $S \models I \models J \models W$.*

Proof. Considering that $(S, \neg W)$ is an unsatisfiable pair of formulas, this theorem is essentially a reformulation of Theorems 3.4 and 3.11. □

This theorem provides a basis for a heuristic in step 3 of the algorithm at the end of Section 4.3.4 (page 34):

We assume that the strongest interpolant $I_1(X_1)$ has been computed in step 1, the interpolant $K_2(X_2)$ has been (inductively) computed before and $C(X_1, X_2)$ is the current rule. Then we compute

$$K_1'(X_1) :\equiv \forall X_2(C(X_1, X_2) \rightarrow K_2(X_2)).$$

We claim that K_1' already fulfills the properties 3 (a) and (b) from page 34. Property 3 (b) is a direct application of Theorem 3.3:

$$K_1' \wedge C \models K_2 \tag{6.2}$$

Moreover, we have

$$\begin{array}{rcll} I_1 \wedge C & \models & I_2 & \text{(tracking property)} \\ & \models & K_2 & \text{(inductively } I_2 \models K_2\text{)}. \end{array}$$

Since Theorem 3.3 also guarantees that K_1' is the weakest formula to satisfy equation (6.2), property 3 (a) follows:

$$I_1 \models K_1'$$

If we set $K_1 := K_1'$, the K_i would exactly be weakest interpolants. However, we want to conciliate between strongest and weakest interpolants and therefore strengthen K_1' to K_1 with Theorem 6.3. That is, we (universally) quantify out the variables that occur in K_1' and not in I_1. Let Z be those variables. Then we set

$$K_1 :\equiv \forall Z.K_1'.$$

Theorem 6.3 yields

$$I_1 \models K_1 \models K_1'.$$

Thus, properties 3 (a) and (b) still hold for K_1.

This heuristic needs to determine which variables "occur" in a formula. When working with BDDs (as we do), this means we need to find out which variables occur as nodes in a given BDD. Fortunately, there are suitable BDD library functions for this task.

Compared to weakest interpolants, this heuristic tends to simplify the predicates. We believe that they get more "meaningful" at the same time. The following example tries to convey this intuition to the reader.

Example 6.4. Consider the spurious trace on the left side of Figure 6.4. To the left of the nodes, the strongest interpolants are displayed; to the right of the nodes, the weakest interpolants are displayed. On the right side of the same figure, the interpolants are computed according to the "conciliating-interpolants" heuristic described above.

For example, when the predicate at node 3 (K_3) is to be computed, the predicate at node 4 has already been computed: $K_4 \equiv X = 0$. First, we compute $K_3' :\equiv (Z \neq 0 \rightarrow X = 0) \equiv (Z = 0 \vee X = 0)$. Then, we determine that Z is a variable occurring in K_3', but not in I_3. Therefore we strengthen K_3' to K_3 by setting $K_3 :\equiv \forall Z(Z = 0 \vee X = 0) \equiv X = 0$.

It can be seen that this heuristic is able to discover the relevant predicate $[X = 0]$, whereas the strongest and the weakest interpolants are obscured by statements that are not relevant for the spuriousness of this trace. We can hope that this predicate is simple and "general" enough to exclude also counterexamples that are not yet contained in the current DAG.

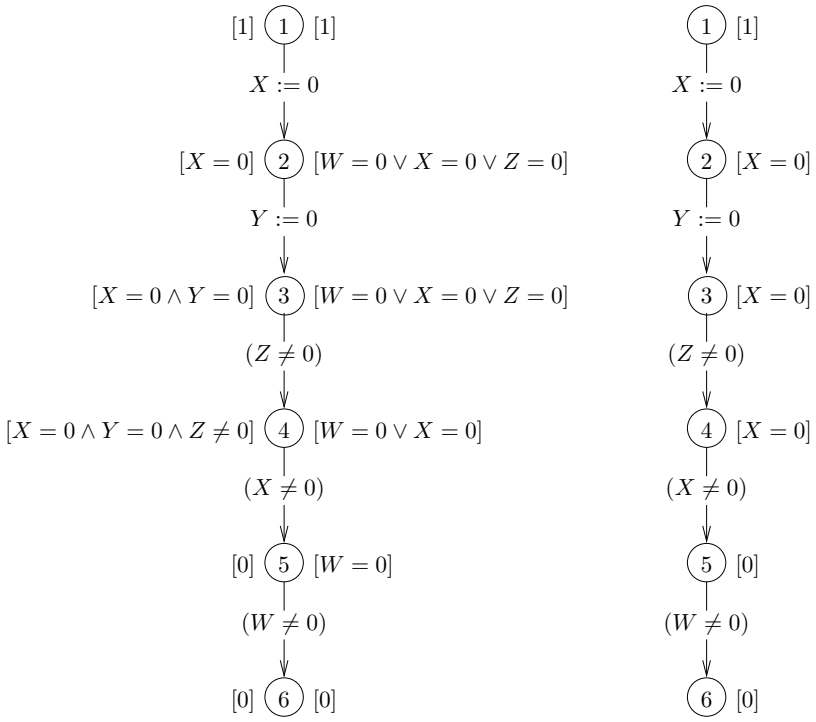

Figure 6.4: Strongest, weakest and conciliated interpolants.

Example 6.5. Consider the procedures in Figure 6.5. A question mark (`?`) stands for a nondeterministic expression. We want to model-check the fact that `error` is not reachable.

The idea of this program is that $X = Y$ is an invariant of this program. The variable Z and the array $Q[\,]$ have, in fact, nothing to do with the reachability of `error`.

We model-checked this program with abstraction using weakest interpolants, with abstraction using conciliated interpolants and with Moped without abstraction. Strongest interpolants performed poorly (not shown). A variable was represented by 3 bits. Figure 6.6 shows the results.

Some variations of the code justify the following interpretation:

- Both the weakest and conciliated interpolants heuristics "notice" that the array $Q[\,]$ is irrelevant. Moped without abstraction does not notice it. Its runtime can be arbitrarily increased by making the array larger. In addition, Moped without abstraction does not significantly benefit from replacing the nondeterministic expression in the while loop by the expression `true`. Abstraction, in contrast, recognizes this infinite loop very quickly (no refinement steps necessary) and concludes that `error` is not reachable.

```
declare X, Y, Z as global

procedure main
     X := 0
     Y := 0
     call foo()
     if X ≠ Y then
          if Z = 1 then
               error

procedure foo()
  declare P as local
  declare array Q[7] as local
     while ? do
          if ? then
               P := ? mod 8
               Q[P] := Q[P] + 1
          if ? then
               X := X + P
               Y := Y + P
               Z := ?
```

Figure 6.5: Code example for conciliated interpolants.

	weakest interp.	conciliated interp.	w/o abstraction
# of CEGAR cycles	5	3	n/a
max. # of predicates	11	6	n/a
time	3.8 sec	0.1 sec	10 sec

Figure 6.6: Results when running code of Figure 6.5.

- Only the conciliated interpolants heuristic "notices" that the variable Z is irrelevant. Whereas the weakest interpolants heuristic gets "confused" by the statements involving Z, the conciliated interpolants heuristic quantifies Z out. This can be seen by removing the nondeterministic assignment of Z: Then both weakest and conciliated interpolants prove the property in less than 0.1 seconds (3 CEGAR cycles, 4 predicates).

Another observation is that conciliated interpolants need less time per CEGAR cycle and also less time per predicate than weakest interpolants. We interpret this as an illustration of our statement above saying that the predicates (and the BDDs representing them) get simpler by quantifying irrelevant variables out.

6.4 Assuming skip as a heuristic

In this section we present another heuristic for the framework presented at the end of Section 4.3.4 (page 34).

It is based on the experience that many statements do not influence the property to be checked and at the same time do not contribute to the spuriousness of the given counterexample DAG. The left side of Figure 6.4 shows an example: Only the statements modifying the variable X matter. Whereas the heuristic in the previous section discovered this fact in a purely semantic way again based on Craig interpolation, one can also detect it more syntactically.

The idea is to compute interpolants bottom-up and assume each statement encountered to be a `skip`-statement unless the strongest interpolant prohibits that. More precisely, this heuristic works as follows.

Speaking in terms of the heuristic framework of Section 4.3.4, we assume that the strongest interpolant $I_1(X_1)$ has been computed in step 1, the interpolant $K_2(X_2)$ has been (inductively) computed before and $C(X_1, X_2)$ is the current rule. Then, $K_1(X_1)$ is to be computed next.

- We first check if setting $K_1(X_1) :\equiv K_2(X_1)$ is valid, i. e., if such a K_1 fulfills the properties 3 (a) and (b) from page 34:

  (a) $I_1 \models K_1$
  (b) $K_1 \wedge C \models K_2$

  If it does, then this K_1 is chosen. It is the same K_1 we would choose in the usual bottom-up computation in the case of $C \equiv$ `skip`.

- Otherwise, we set

  $$K_1(X_1) :\equiv \forall X_2(C \rightarrow K_2).$$

  This K_1 fulfills properties 3 (a) and (b), as can be proved inductively as in Section 6.3.

Example 6.6. Consider again Figure 6.4. The predicates on the very left of this figure are strongest interpolants. We claim that the predicates on the very right of this figure are interpolants computed according to the heuristic described in this section.

For example, when the predicate at node 4 (K_4) is to be computed, the predicate at node 5 has already been computed: $K_5 \equiv 0$. First, it is checked if K_5 can be copied to K_4, i. e., $K_4 \equiv 0$. This cannot be done, because 0 is not implied by the strongest interpolant $I_4 \equiv (X = 0 \wedge Y = 0 \wedge Z \neq 0)$. Therefore, we have to set $K_4 :\equiv (X \neq 0 \rightarrow 0) \equiv (X = 0)$.

In contrast, when we compute K_3, we can copy $K_4 \equiv X = 0$, because both

$$X = 0 \wedge Y = 0 \models X = 0 \qquad \text{and}$$
$$X = 0 \wedge C \neq 0 \models X = 0$$

hold.

A nice feature of this heuristic is that it can omit whole procedures in the refinement step. The idea is that we can suspect that a procedure does nothing important for the property to be checked.

Example 6.7. Consider the program in Figure 6.7. It is clear that the reachability of `error` does not depend on `foo`, because `foo` cannot modify `main`'s local variable B.

```
procedure main
  declare B as local
m0:  B := 0
m1:  call foo()
m2:  if B ≠ 0 then
          error
```

```
procedure foo()

f0:  do something very
     complicated possibly
     involving global
     variables
```

Figure 6.7: Procedure `foo` is irrelevant.

Model-checking an abstract version of this program and computing interpolants according to this heuristic might yield a counterexample DAG as sketched in Figure 6.8.

The point is that the first refinement iteration is the only one. In particular, refinement of the procedure `foo` is not necessary. The predicate $[B = 0]$ at the "pivot-point" $f_0 m_2$ implies the predicate $[B = 0]$ at node m_2, regardless of what `foo` may do with the global variables. Clearly, arbitrary savings are possible with this abstraction heuristic.

On a more philosophical level, one could argue that this heuristic exploits the abstraction provided by the programmer. By giving the program a procedural structure, the programmer states explicitly that the program can be decomposed into several subtasks. Our heuristic can possibly identify the subtasks relevant to the property. This is the case in Example 6.7. The next section gives further evidence.

6.5 Combining multiple heuristics

We think that there are more heuristics yet to be discovered. Other heuristic ("meta-heuristics") can be obtained by combining different heuristics.

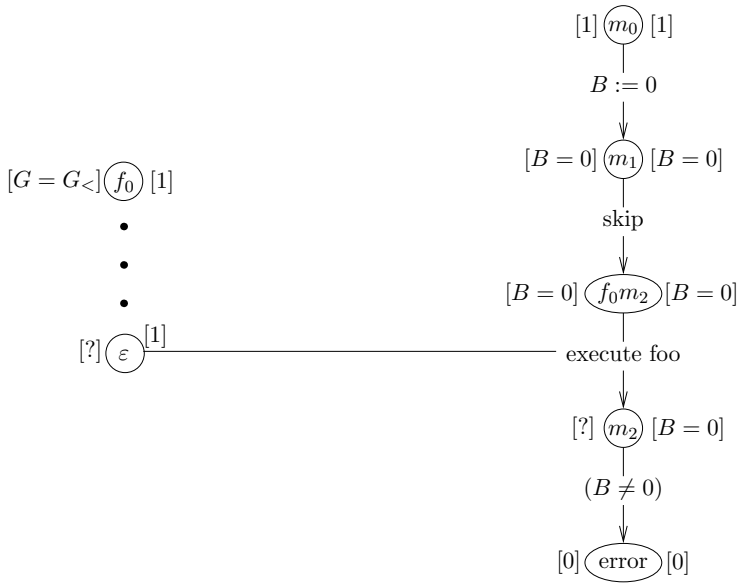

Figure 6.8: A counterexample DAG marked with interpolants.

Different heuristics generally do not exclude each other. One could simply add predicates according to several heuristics. By doing that, one might very well produce more predicates than necessary to rule out the current counterexample DAG. This can be tolerated though if the predicates pay off, e. g., lead to fewer refinement cycles.

However, one can closely integrate the heuristics from Sections 6.3 and 6.4 as follows: When moving from K_2 to K_1 in the bottom-up computation, one first applies the heuristic from Section 6.4 ("assuming skip"). Call the obtained predicate K_1'. To get the final K_1, we "conciliate" between the strongest interpolant K_1' and I_1, as described in Section 6.3.

Example 6.8. Figure 6.9 sketches code for a program involving quicksort. We want to model-check the fact that `error` is not reachable. This does not depend on the quicksort part. The procedure `one()` returns the number 1 by assigning it to the variable G. We included this procedure in order to show that our heuristics do not blindly abstract away procedure calls.

We used, in turn, the "assuming skip" heuristic from Section 6.4, the combined heuristic described above and Moped without abstraction. Other heuristics did not perform as well. Figure 6.10 shows the results.

The combined heuristic clearly outperforms the others. It identifies the relevant procedure `one()` and does not refine the quicksort procedure.

```
declare G as global
declare arrays X[4], Y[4] as global

procedure main
  declare B as local
     call quicksort() such that it sorts X[ ]
     call one()
     B := G
     call quicksort() such that it sorts Y[ ]
     if B ≠ 1 then
          error

procedure quicksort()
  ⋮

procedure one()
     G := 1
```

Figure 6.9: Code example for combined heuristics.

	assuming skip	assuming skip + conciliated	w/o abstraction
# of CEGAR cycles	4	1	n/a
max. # of predicates	9	3	n/a
time	3.5 sec	0.1 sec	1.7 sec

Figure 6.10: Results when running code of Figure 6.9.

6.6 An application to a Java program

This section gives further evidence of the practical usefulness of the concepts developed in this thesis.

With the tool jMoped [SSE05], we can translate Java Bytecode into Moped's input language. Using the Java compiler as a front-end, we are thus able to test our CEGAR scheme on real Java programs. They need to fulfill some requirements that are listed in [SSE05]. In particular, the number of objects must be known beforehand.

We found suitable code in the book [Wei98], which is a textbook about data structures and algorithms in Java. We chose a program for probabilistic primality testing, see Figure 6.11.

The code in this program is slightly modified from the version in the book:

- We inserted an assertion (`n != 0`) at the beginning of the (recursive)

procedure `witness`. This check is motivated by the fact that in the same procedure a modulo division by `n` is carried out. We want to check with our abstraction scheme that this assertion holds.

- We replaced a call of a random-number generator in the procedure `isPrime` by the expression `Integer.parseInt(h)`. The tool jMoped automatically abstracts this library call and leaves the value of this expression nondeterministic. Thus, the random-number generator is simulated. Strictly spoken, the random-number generator is overapproximated. If the property holds in this program, then it must also hold in the original program that uses the random-number generator.

The assertion to be checked is, in fact, valid: The positive numbers in the loop of the main procedure are propagated to the procedure `witness` via the parameter `n`. This procedure, in turn, calls itself passing again the same parameter `n`. Since `n` is the only number occurring in the denominator of a division, a division-by-zero error cannot occur.

We compiled this program with the Java compiler and used jMoped with its default options, i. e., 4 bits per integer variable. Our CEGAR scheme proves in three refinement cycles that the assertion holds, producing at most two predicates per control point. The runtime is about 0.5 seconds. This is already the case for the weakest-interpolants heuristic. The other heuristics introduced in this chapter perform very similarly.

Moped without abstraction, in contrast, does not benefit from the fact that the assertion is quite simple. Because of the nondeterminism of this probabilistic algorithm, the state space is large. Moped without abstraction needs around 9 seconds in this example to verify the assertion. By increasing the number of bits, the performance difference between Moped with and without abstraction can be further increased.

```
1  public class Fig10_62
2  {
3      /**
4       * Method that implements the basic primality test.
5       * If witness does not return 1, n is definitely composite.
6       * Do this by computing a^i (mod n) and looking for
7       * nontrivial square roots of 1 along the way.
8       */
9      private static int witness( int a, int i, int n )
10     {
11         assert (n!=0);
12         if( i == 0 )
13             return 1;
14
15         int x = witness( a, i / 2, n );
16         if( x == 0 )    // If n is recursively composite, stop
17             return 0;
18
19         // n is not prime if we find
20         // a nontrivial square root of 1
21         int y = ( x * x ) % n;
22         if( y == 1 && x != 1 && x != n - 1 )
23             return 0;
24
25         if( i % 2 != 0 )
26             y = ( a * y ) % n;
27
28         return y;
29     }
30
31     /**
32      * The number of witnesses queried in
33      * randomized primality test.
34      */
35     public static final int TRIALS = 5;
36
37     /**
38      * Randomized primality test.
39      * Adjust TRIALS to increase confidence level.
40      * @param n the number to test.
41      * @return if false, n is definitely not prime.
42      *     If true, n is probably prime.
43      */
44     public static boolean isPrime( int n )
45     {
46         for( int counter = 0; counter < TRIALS; counter++ ) {
47             String h = new String();
48             // Integer.parseInt(h) will return a
49             // nondeterministic value in jMoped's translation
50             if( witness( Integer.parseInt(h), n - 1, n ) != 1 )
51                 return false;
52         }
53
54         return true;
55     }
56
57     public static void main( String [ ] args )
58     {
59         for( int i = 3; i < 10; i += 1 )
60             if( isPrime( i ) )
61                 System.out.println( i + "␣is␣prime" );
62     }
63 }
```

Figure 6.11: A probabilistic primality testing algorithm

Chapter 7

Implementation aspects

7.1 Succinct representation of the predicates

Although, at first glance, Example 4.1 looks as if we need one bit per predicate (three, in this example) in the abstract program, we rather need less. Because the predicates are localized, we need only as many bits as the maximum number of predicates at some control point.

In Example 4.1, there are two control points: One with one predicate, the other with two. We need just two bits, because l_1 and l_3 are not used at the same time and we could identify their bits. Doing this, the abstracted version of the instruction would look as simple as $l_2 := 1$. (Keeping the value of $l_1 = l_3$ is implicit.)

On the other hand, the abstract version of an instruction is some relation over the boolean variables involving the *new* and the *old* bit values. In Example 4.1, for instance, the bits of l_1 and l_2 could be shared instead of l_1 and l_3. In the abstract program, the new value of $l_1 = l_2$ would be 1, and the new value of l_3 would be the *old* value of $l_1 = l_2$.

As described in Section 2.1, the rules of a symbolic PDS describe relations between the old and the new values of the program variables. Therefore, copies of the program variables are needed anyway. In the case of a procedure call, another copy is needed. Since Moped produces another copy for internal purposes, the number of BDD nodes roughly equals four times the number of bits in the program variables.

7.2 Incremental concretization

In Section 5.3.1 we stated how to build formulas describing the correspondence between the abstract variables and their meanings. They are of the

form

$$\{n\} \equiv (l_1 \leftrightarrow P_1) \wedge \cdots \wedge (l_m \leftrightarrow P_m),$$

where the l_i are abstract variables and the P_i their meaning, i. e., some predicates over concrete variables.

In the course of the CEGAR loop (see Chapter 4), additional predicates will be obtained that are to be added to the abstraction. For this purpose, we keep lists of predicates for each control point. Those lists are extended whenever a new predicate is derived.

For efficiency, we apply a similar strategy to the concretizations: If a new predicate, say P_{m+1}, is derived, we *update* the concretization $\{n\}$ as follows:

$$\{n\} :\equiv \{n\} \wedge (l_{m+1} \leftrightarrow P_{m+1})$$

Thus, the work done before need not be redone.

7.3 BDD variable ordering

The BDD variable ordering can have a significant influence on the size of BDDs [Bry86]. Therefore one should care about an appropriate ordering of the variables (see e. g. [Sch02]).

Besides the size of the BDDs, the typical operations also matter. It turned out in our experiments that the quantification operations form a bottleneck. Existential quantification is performed in two stages of the CEGAR loop:

- construction of the abstract program using existential abstraction (see Sections 4.1 and 5.3),
- predicate generation via computing interpolants (see Section 4.3).

In both cases, we quantify over concrete variables. Considering the structure of BDDs, it seems advantageous to quantify over variables that occur "late" in the variable order, i. e., occur at the bottom of the BDDs.

Therefore, we chose to order the abstract variables *before* the concrete variables. Thus, the costs for quantification over concrete variables are lowered.

Experiments validated this intuition, i. e., runtime was clearly smaller with this variable order. We have not yet tried any further fine-tuning of the variable order.

7.4 The DAG representation

Given a DAG of counterexamples, we want to compute interpolants in a top-down or in a bottom-up way, as described in Sections 4.3.2 and 4.3.3.

Of course, we want to avoid computing the predicates twice for the same node. As a consequence, we need a data structure for DAGs that records for each node if its interpolant has already been computed.

Another requirement for the data structure is that each node can access its neighbor nodes in constant time, along with their interpolants (if already computed). We ask for this, because for computing the interpolant of a node, we need in general the interpolants of the neighbors.

Both requirements lead us to a data structure closely related to the pictures of DAGs that we have drawn in this thesis. Each node is represented as an object in memory with the following attributes in particular:

- a list of "up-edges", i. e., edges whose target is the current node,
- a list of "down-edges", i. e., edges whose source is the current node,
- a field storing the already computed interpolants.

This enables us to carry out the top-down computation in a recursive *dynamic-programming* fashion according to Section 3.3.3. Figure 7.1 reformulates Section 3.3.3 algorithmically. The computation of *all* strongest interpolants can be triggered by calling getI($n_\bot$).

```
procedure getI(node n) returns predicate over X_n
/*returns the strongest interpolant I_n(X_n)*/
if I_n has already been computed then
        return I_n
if n = n_⊤ then
        I_n := 1
        return I_n
I_n := 0
forall upedges e = (n', n) do
        I_n'(X_n') :≡ getI(n')
        I_n :≡ I_n ∨ ∃X_n'(I_n'(X_n') ∧ F_e(X_n', X_n))
return I_n
```

Figure 7.1: Recursive top-down computation.

There is corresponding algorithm for a recursive weakest interpolants computation (not shown).

In the case of procedures, the DAGs have more complicated structures (see Chapter 5). However, the idea remains the same: The graph is represented by a pointer structure, and the already computed interpolants are saved in order to avoid recomputation.

Chapter 8

Future work

8.1 Heuristics for the model-checking step

In Chapter 6 we described heuristics for the choice of interpolants. Whereas those heuristics pertain to the predicate-generation step, we could also try to "tweak" the model checker in the sense that we try to make it report counterexamples that are "suitable" for the predicate-generation step. It might be necessary to change some internal strategies of Moped for such experiments.

8.1.1 Moped's search process

The core of Moped is its search for reachable configurations. Moped's default behavior is to abort this search as soon as an error configuration has been found reachable.

When using Moped in the model-checking step of our CEGAR scheme, we currently force Moped to perform a complete search in the sense that an automaton is constructed that accepts *all* reachable configurations.

The rationale for this policy is our hope to get "big" DAGs, i. e., many counterexample traces in a single DAG. Thus, we expect to obtain more meaningful interpolants. At the same time, the model-checking step is not a bottleneck. Therefore, we can afford putting some effort into this step.

On the other hand, this policy may sometimes lead to long counterexamples, thus producing much work in the predicate-generation step.

A possible enhancement might be to employ some sort of breadth-first search in Moped's search step. The search could be stopped as soon as the error label is reached. This might lead to "thick" but short DAGs, i. e., many short counterexamples in a single DAG, which seems to be desirable.

8.1.2 Less recursive calls

An error configuration is a configuration with an error label on top of the stack. When Moped has completed its search, it looks for an error configuration that is accepted by its automaton of the reachable configurations. Especially in the case of recursive programs, there might be multiple such configurations.

Currently, Moped tends to pick very long configurations with many recursive calls on the stack. This seems not desirable for our CEGAR purposes. We believe that a breadth-first search that looks for short error configurations (i. e., configurations with few calls on the stack) could clearly improve performance in many cases of recursive programs.

8.2 Generalization to LTL

Moped is, in principle, not restricted to reachability checks. Rather, it can handle full LTL specifications. [Sch02] describes this in detail, we only sketch the process here. Let $\mathcal{P}$ be the symbolic PDS to be model-checked and φ the LTL specification.

First, a Büchi automaton for $\neg\varphi$ is constructed, i. e., a Büchi automaton $\mathcal{B}$ with $L(\mathcal{B}) = L(\neg\varphi)$. Then, a product automaton $\mathcal{BP}$ of $\mathcal{B}$ and $\mathcal{P}$ is computed. If $\mathcal{BP}$ has an accepting run, then this run is a counterexample, i. e., φ does not hold for all computations of $\mathcal{P}$.

Note that a counterexample in LTL is, in general, infinite. It can be given as a prefix α and a loop β. I. e., a counterexample looks like $\alpha\beta\beta\beta\cdots$, where α and β are finite sequences of control points. In addition, as in the reachability case, the rules for transforming a control point into its successor are given.

We can use such counterexamples for a CEGAR scheme in a similar way as described in this thesis. The detection of spurious counterexamples is different though, because if a counterexample is spurious, it is not obvious how many iterations of β may be needed until the counterexample gets spurious.

For now, assume that we have a monolithic program without procedure calls, a finite set X of boolean variables and a single counterexample $c = \alpha\beta\beta\beta\cdots$ [CGJ+00]. In order to check for spuriousness, we first compute the effect of α, i. e., the possible variable values after the execution of α. This can be given by some predicate $P(X_0)$.

On the other hand, the effect of β can be formulated as a relation of the variables before execution of β and the variables after execution of β. Call this relation $R(X, X')$.

The question whether c is spurious boils down to the question if

$$F :\equiv P(X_0) \wedge R(X_0, X_1) \wedge R(X_1, X_2) \wedge R(X_2, X_3) \wedge \cdots$$

is unsatisfiable. Define

$$R^n(X_0, X_n) :\equiv \exists X_1 \cdots \exists X_{n-1} \bigwedge_{i=0}^{n-1} R(X_i, X_{i+1}).$$

It is easy to see that F is unsatisfiable if and only if there is some n such that

$$F'_n(X_n) :\equiv \exists X_0 (P(X_0) \wedge R^n(X_0, X_n))$$

is unsatisfiable.[1] Consider the sequence

$$F'_1(X), F'_2(X), \ldots$$

There can only be $2^{|X|}$ non-equivalent formulas in this sequence. In addition, if $F'_i \equiv 0$, then $F'_{i+1} \equiv 0$. We can therefore conclude:

$$\begin{aligned}
\text{c is spurious} &\iff F \text{ is unsatisfiable} \\
&\iff F'_n \text{ is unsatisfiable for some } n \\
&\iff F'_n \text{ is unsatisfiable for some } n \leq 2^{|X|} \\
&\iff F'_n \text{ is unsatisfiable for } n = 2^{|X|}
\end{aligned}$$

Thus, in order to check for spuriousness, one essentially has to compute $R^{2^{|X|}}$. Fortunately, this can be carried out by *repeated squaring*:

$$R^{2n}(X_0, X_{2n}) \equiv \exists X_n (R^n(X_0, X_n) \wedge R^n(X_n, X_{2n}))$$

In the spurious case, we can also use repeated squaring to determine the *shortest spurious prefix* of c. It is essentially given by the smallest i such that F'_i is unsatisfiable. This i can be found with a binary search and by combining the previously computed R^{2^j} accordingly.

In conclusion, we can transform an infinite spurious counterexample c into a finite one. Starting from a finite counterexample, we could essentially apply the same methods for predicate generation as described in this thesis.

We suspect that in practice, spurious counterexamples tend to be short (very much shorter than $2^{|X|}$). We think, we can generalize those ideas to counterexample DAGs and to general symbolic PDSs.

[1] One could use the compactness theorem of propositional logic to prove that.

8.3 Lazy Abstraction

In [HJMS02], a CEGAR scheme called *Lazy Abstraction* is presented that integrates the steps of the CEGAR loop in Figure 1.1. It aims at two sorts of savings:

- "Explore" only the parts of the state space that have not yet been shown to be error-free.
- Refine each part of the state space only as much as required.

In our CEGAR scheme, there is currently no need for savings of the first sort, since the model-checking step does not constitute a bottleneck. However, the second kind of saving could help us tackle bottlenecks in the predicate-generation and abstraction steps.

We sketch here how an adaption of *Lazy Abstraction* to our approach could look like.

The first idea for a saving is that the abstraction of a rule has to be recomputed only if the control points that the rule involves have been assigned additional predicates. That means, it might be fruitful to keep track of the control points with new predicates and only recompute their part of the abstraction.

In this spirit, we could save work in the abstraction step by stopping predicate-generation as soon as possible. As a desired side effect, we would invest less effort into computing predicates.

The obvious question is how to know when to stop predicate-generation. The idea is to use the information about the abstract variable values that Moped produces in the model-checking step. Figure 4.5 shows an example of a DAG including the abstract variable values. We stated in Section 4.3.1 that we do not use those abstract values, but rather focus on the "raw" DAG that describes the control flow. In contrast, lazy abstraction utilizes those abstract values, more precisely: their concretization.

We illustrate that by means of a simple example.

Example 8.1. Consider the code in Figure 8.1. The code between line 2 and 11 does not modify X.

We assume that we have already derived the predicate $[X = 0]$ (or its negation, which makes no real difference) in a previous abstraction cycle. It is localized at the control points 2 and 11 and was probably added in order to exclude some spurious error in the procedure `foo`. In the next cycle, Moped might give us another spurious error trace as shown in Figure 8.2.

For each node, Moped computes a predicate describing the possible abstract variable values. In the figure, to the left of the nodes, we have already re-

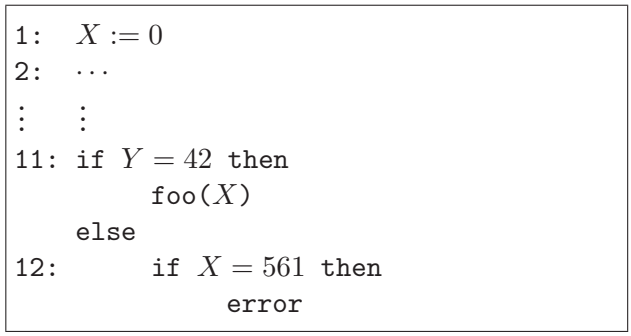

Figure 8.1: Code example for lazy abstraction.

placed those abstract variable values by their concretizations. For instance, $[[X = 0]]$ to the left of node 11 means that the abstract variable corresponding to $[X = 0]$ is set to 1 at node 11. At the error node, $[[1]]$ means that nothing is known about the abstract variables at that point.

To the right of the nodes, weakest interpolants, computed in the bottom-up way, are displayed. Computation of weakest interpolants can be stopped as soon as the concretization on the left side *implies* the interpolant on the right side. In Figure 8.2, this is the case at node 11, where $[X = 0]$ implies $[Y = 42 \ \vee X \neq 561]$. As illustrated (a little drastically with the big "X"), no further interpolants have to be computed above node 11.

We claim that $[X \neq 561]$ at node 12 is the only new predicate needed to exclude the counterexample of Figure 8.2. To see that, notice that, by construction of the weakest interpolants at nodes 11 and 12 (tracking property), we have

$$[Y = 42 \ \vee X \neq 561] \ \wedge \ [Y \neq 42] \quad \models \quad [X \neq 561].$$

Since the concretization on the left side of node 11 $[X = 0]$ implies the weakest interpolant, we also have

$$[X = 0] \ \wedge \ [Y \neq 42] \quad \models \quad [X \neq 561].$$

The next abstract program, which takes those predicates into account, will therefore not allow for a rule between control points 11 and 12, where $[X = 0]$ at node 11 is set to 1, but $[X \neq 561]$ at node 12 is set to 0. Once the interpolant at node 12 $[X \neq 561]$ is set to 1, the rest of the trace is not feasible. Thus, the current counterexample is excluded.

The example shows how predicate generation can be shortened by taking advantage of previously found predicates. This form of lazy abstraction might lead to substantial improvements.

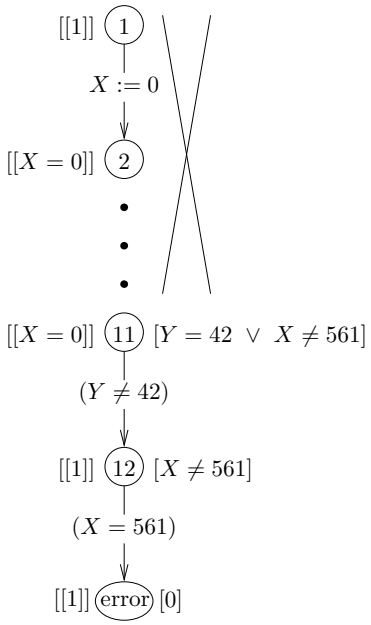

Figure 8.2: Spurious trace with abstract and concrete predicates.

8.4 Program slicing

We mentioned slicing already in Section 1.4: Pre-compute the set of variables without a potential influence on the property to be checked and remove all statements only modifying those variables. Slicing is orthogonal to our method: The result of the slicing step would be a new (and hopefully simpler) symbolic PDS that, in turn, could be model-checked by the CEGAR scheme described in this thesis.

8.5 Other program and predicate representations

We have presented our CEGAR scheme while emphasizing that BDDs can be used throughout the CEGAR loop. However, this is actually not absolutely necessary for our scheme. For the concrete program to be model-checked and for the predicate representation, other data structures could be used as well.

Especially when quantifying over concrete variables that are represented by many bits, our CEGAR scheme with BDDs gets computationally expensive.

This raises the question in which cases other means of set representation may be more appropriate. We can currently think of mainly two alternative representations that are, in principle, able to represent infinite sets:

- Symbolic representation. This is the most commonly used representation in CEGAR. In particular, the SLAM project [BR01] and the BLAST tool [HJMS02] use symbolic forms of representation. A set is described by an arithmetic constraint. Theorem provers (such as Simplify [DNS00]) are needed to detect relations between different sets. In the context of CEGAR, theorem provers are used particularly in the predicate-generation step.

- Representation as NDDs [WB95]. NDDs can represent sets of vectors of integers, including all Presburger-definable sets. An NDD is essentially a finite automaton that accepts a set of integer vectors. NDDs have been suggested for model checking in [WB98]. A toolkit for possible model-checking implementations is available [BFL01].

The idea would be to use Moped for model-checking the abstract system. That means that the abstract system would still be given as a symbolic PDS with BDDs. The concretization and the predicates, however, would be based on different data structures. Thus, we could handle another "dimension of infinity" with Moped. However, we would have to accept the fundamental restrictions of theorem provers (see Section 2.4.2).

Chapter 9

Conclusions

We have successfully designed and implemented a CEGAR scheme for pushdown systems.

We developed a theoretical framework based on Craig interpolation for propositional logic and applied it to the abstraction of programs given by BDDs. It turned out that this theory reveals a wide field of possible heuristics for deriving relevant predicates. Our interpolant framework might give insights in the properties of meaningful predicates and their computation.

We generalized the theoretical framework to symbolic pushdown systems, which model sequential programs and can define infinite state spaces.

In our implementation, we tightly integrated the model checker Moped in our program. In particular, we used internal data structures of Moped in order to obtain whole DAGs of counterexamples in a single run of Moped. A natural extension of our interpolant theory enables us to exclude those DAGs in a single cycle of the CEGAR loop.

Our experiments showed that our CEGAR scheme augments the set of programs that can be model-checked with Moped. However, there were also instances that seemed not suitable for abstraction and should be model-checked directly. We hope to find concrete evidence that many real-world examples benefit from our abstraction.

Only few instances performed similarly when running Moped with and without abstraction. Typically, one version was clearly superior to the other. A general criterion that separates those two classes is hard to give. If the property to be checked is simple and the program complicated, then abstraction often pays off. A conclusion is that several versions (including different heuristics) should be taken into account when using Moped for model checking.

The use of BDDs in both the concrete and the abstract domains was a

central theme of our work. BDDs fit nicely in our framework, however, their use can get costly when large amounts of data are involved. Whereas in other projects, the calls of theorem provers constitute the bottleneck, we experienced that quantifications over the concrete variables dominates the computation time. We conclude that, in both paradigms, the predicate-generation and abstraction phases need to be optimized.

We think that there are various possibilities for future enhancements of our scheme. As outlined in Chapter 8, this concerns all phases of the CEGAR loop, especially the predicate generation. We believe that meaningful predicates are the key for efficient CEGAR schemes, because they lead to few refinement cycles. We are also thinking about alternative forms of data representation.

In conclusion of this thesis, we think that the limits of abstraction refinement have not yet been reached. Future research in this area should reveal its full power.

Bibliography

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR'97*, LNCS 1243, pages 135–150, 1997.

[BFL01] B. Boigelot, J.-M. François, and L. Latour. The Liège Automata-based Symbolic Handler (LASH), 2001. Beta version available from `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[BR00] T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer, 2000.

[BR01] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: Workshop on Model Checking of Software*, LNCS 2057, pages 103–122, 2001.

[BR02] T. Ball and S.K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, 2002.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CCG+03] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE Computer Society Press, 2003.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV'00*, LNCS 1855, pages 154–169. Springer, 2000.

[CGP02] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 2002.

[CKSY04] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.

[CKSY05] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proceedings of TACAS 2005* [TAC05], pages 570–574.

[Cra57] W. Craig. Linear reasoning. A new form of the Herbrand-Genzen theorem. *Journal of Symbolic Logic*, 22:250–268, 1957.

[DH99] M.B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proc. ACM Workshop on Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[DNS00] D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover, 2000. `http://research.compaq.com/SRC/esc/Simplify.html`.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. CAV'97*, LNCS 1254, pages 72–83. Springer, 1997.

[HJMM04] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Proc. POPL'04*, pages 232–244. ACM Press, 2004.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL'02*, pages 58–70. ACM Press, 2002.

[McM03] K.L. McMillan. Interpolation and SAT-based Model Checking. In *Proc. CAV'03*, LNCS 2725, pages 1–13. Springer, 2003.

[McM05] K.L. McMillan. Applications of Craig interpolants in model checking. In *Proceedings of TACAS 2005* [TAC05], pages 1–12.

[RSJMar] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, To appear.

[Sch02] S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, TU-München, 2002.

[Som98] F. Somenzi. Colorado University Decision Diagram Package, 1998. `ftp://vlsi.colorado.edu/pub`.

[SSE05] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *Proceedings of TACAS 2005* [TAC05], pages 541–545.

[TAC05] *Proceedings of TACAS 2005*, LNCS 3440. Springer, 2005.

[WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 21–32. Springer, 1995.

[WB98] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. CAV'98*, LNCS 1427, pages 88–97. Springer, 1998.

[Wei98] M.A. Weiss. *Data Structures and Algorithm Analysis in Java.* Addison-Wesley, 1998.