An Abstraction/Refinement Scheme for Model Checking C Programs

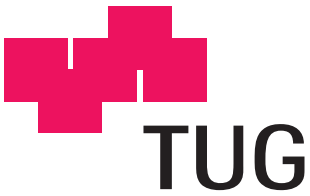
Georg Weißenbacher

März 2003

Diplomarbeit in Telematik

durchgeführt am

IST - Institut für Softwaretechnologie
der Technischen Universität Graz



Betreuer: Dr. Roderick Bloem
Begutachter: Univ.Prof. DI Dr. Franz Wotawa

Ich versichere, diese Arbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubter Hilfsmittel bedient zu haben.

Abstract

We present an approach for automatic software verification that combines the benefits of the two complementary concepts of theorem proving and model checking. We apply theorem proving and predicate transformers to remove detail from a C program to such an extent that a model checker can be applied to search for erroneous execution traces (with respect to a specification that must be provided by the user). The existence of such an error trace in the abstract model indicates either an error in the original C program or it indicates that too much detail was omitted. In the former case the corresponding error trace in the C program is reported; In the latter case, we refine the abstraction by adding more detail and reiterate the process. If the model checker is unable to find an error trace then the C program is correct with respect to the specification.

We will give a detailed description of this approach and show how it can be used to detect errors in C Programs. We will furthermore discuss the limits of the approach.

Zusammenfassung

Diese Arbeit behandelt einen automatisierten Ansatz zum Auffinden von Pfaden in C Programmen, die bezüglich einer vom Benutzer vorgegebenen Spezifikation zu einem Fehler in der Ausführung des Programmes führen. Die Validierung des Programmes erfolgt statisch, d.h. es müssen keine Testfälle vorgegeben werden. Mit Hilfe eines Theorem Beweisers und Prädikatenabstraktion wird das C Programm so weit vereinfacht, dass ein Model Checker verwendet werden kann, um nach fehlerhaften Pfaden zu suchen. Die Existenz solcher Pfade im vereinfachten Programm weisst entweder auf einen Fehler im ursprünglichen Programm hin, oder darauf, dass das Programm zu stark vereinfacht wurde. Im ersteren Fall wird der Fehler dem Benutzer gemeldet, während im letzteren Fall ein weniger stark vereinfachtes Programm generiert und neuerlich überprüft wird. Sollte kein fehlerhafter Pfad gefunden werden, so erfüllt das ursprüngliche Programm die vorgegebenen Eingeschaften.

Diese Diplomarbeit enthält eine detaillierte Beschreibung dieses Ansatzes und zeigt, wie er zur Suche von Fehlern in C Programmen verwendet werden kann. Weiters wird auf die Beschränkungen des Ansatzes eingegangen.

Acknowledgments

I would like to express my deep gratitude to Roderick Bloem, whose support went far beyond the usual supervision.

I also want to thank Konrad Slind, for his detailed answers to my questions regarding the HOL theorem prover, Stefan Schwoon, for his help with the MOPED model checking tool, and Thomas Ball, for helping me to understand the SLAM abstraction algorithm. I was surprised how supportive most of the scientists of other research teams were.

Additionally, I want to thank Peter Lucas, Bernhard Aichernig, and Peter Gorm Larsen for directing my research interests towards formal methods. Throughout my studies, they strengthened my belief in verification and formal methods.

Special thanks go to my parents for supporting me all those years. They were the ones who made this thesis possible.

Contents

Chapter 1

Introduction

While devices that are equipped with software driven embedded processors have become an indispensable part of our everyday life we have still not found a way to guarantee that these systems "behave correctly" in every situation. Automatic verification of software and hardware poses a problem that has been investigated by computer scientists for decades. While there have been remarkable achievements in hardware verification in the last few years the results for software are still discouraging. However, the necessity for software verification is growing, since customers are unlikely to accept their cellular phones and washing machines to be as unreliable as desktop computers are nowadays. Developers of safety critical applications are already aware of this necessity, and for them the use of formal methods is obligatory. However, it is still a long road to go until verification tools become sophisticated enough to be accepted by the average programmer. In this thesis we present an approach for automatic software verification that is promising to fill this gap.

Verification of large amounts of code can only be accomplished with help of a *push-button* tool that does not require much user interaction. Model checking is a fully automatic approach that provides explanations for the inadequacy of the inspected model if an error is detected. However, model checking can only be applied to relatively small systems that adhere to certain restrictions that we will discuss later. Theorem proving, on the other hand, can be applied to highly complex systems, but results can only be achieved with extensive manual guidance by a skilled computer scientist. Furthermore, proof based approaches usually require an annotated model. The approach that we present here combines the benefits of these two complementary concepts. With help of a theorem prover we cut down the size of the state space of a C program so that model checking can be applied to the resulting abstraction.

Since we cannot assume that the piece of code we investigate contains annotations or invariants we need to provide a number of constraints that must not be violated. Such a constraint might be that a process in an operating system is not permitted to use a device unless it has been granted access to it. Such *temporal safety properties* can be expressed by specifying program points (resp. labels) that can only be reached if the property is violated. The functions of a program must assert the specified property whenever they are called and jump to a specific label whose reachability signals an error in the implementation if the assertion does not hold. Later in this thesis we will explain how this method can be used to ensure certain properties of a Linux device driver.

Given a C program and a point in this program (specified by a label) we wish to check if this point is reachable, and if this is the case, we demand a detailed explanation of how it can

be reached (resp. an execution path that leads to the label). We have implemented a tool called BOOP that is capable of providing this information. The BOOP toolkit is essentially a reimplementation of the tools C2BP and NEWTON that are part of the Microsoft SLAM toolkit[1]. While the SLAM toolkit provides its own model checker BEBOP for model checking Boolean Programs [BR00] we have integrated the model checker MOPED[2] into the BOOP toolkit. The abstraction algorithm that we present in Chapter 4 is the same as presented in [BMMR01]. We present a more elegant and evident way to find new predicates to refine the abstraction and compare it to the approach that was used for NEWTON and presented in [BR02]. Furthermore we show that there are classes of programs that neither the SLAM nor the BOOP toolkit can handle efficiently.

The algorithm we implemented involves the following three steps:

1. *Obtain an abstraction by removing detail from the original C program.* This is accomplished by modeling the impacts of C statements on the control flow by means of *predicates* over the infinite state of the C program. This yields an abstraction where statements of which we assume that they have no influence on a potential path to the program point in question are ignored. The effects that the remaining statements have on the control flow are modeled by *Boolean Programs*. Boolean Programs are similar to C programs with the restriction that all variables have Boolean type. Each Boolean variable in the abstraction corresponds to a predicate that represents an assumption about the state of the C program. At each point in the Boolean Program, this assumption can either be true or false. Each C statement potentially changes the truth value of this assumption. This is reflected by an assignment statement at the corresponding program point in the Boolean Program. BOOP uses the theorem prover HOL98[3] to determine how the truth values of predicates are affected by C statements.

2. *Model checking the abstraction.* With help of the model checking tool MOPED we try to find an execution path in the abstract model that leads to a violation of the given safety property. If no such path exists then the C program is correct with respect to the given property. The existence of an erroneous execution path indicates either a violation of the property in the original C program, or that too much detail was removed in the abstraction phase. If the path can be shown to be feasible in the original C program, it is reported to the user.

3. *Finding new predicates to refine the abstraction.* If the model checker reports an error path that we found not to be feasible in the original C program, we omitted too much detail when generating the abstraction. We use the information that the model checking tool supplies to discover additional predicates that can be used to refine the abstraction to such an extent that the spurious error path is eliminated.

These steps are repeated until the tool finds a path to the specified label or until it finds out that the label is unreachable. In general this may result in a great number of iterations. Even worse, the algorithm is potentially non-terminating for certain C programs. We will show in Section 7.4 that there is a class of programs for which the algorithm is practically infeasible. However, we found that in many cases the approach yields a result after a few iterations.

[1] `http://research.microsoft.com/SLAM`
[2] `http://wwwbrauer.in.tum.de/~schwoon/moped/`
[3] `http://hol.sourceforge.net/`

A potential target for the method we present are Open Source projects. Projects like the Linux Kernel[4] involve many programmers that contribute their code via the Internet. This "bazaar"-model [Ray98] of development imposes many challenges. The stability of the Linux Kernel is to a great part due to the tremendous number of people who sacrifice each minute of their spare time to test and improve the kernel. As Linux is becoming more popular more and more hardware manufacturers provide kernel drivers for their devices. The impact of these contributions to the monolithic Linux Kernel cannot be estimated yet. However, integration testing of these components will definitely become more costly and less effective as their number grows. A specification of temporal safety properties that device drivers must adhere to could be given by writing stubs for the kernel functions that jump to a specific label if one of the demanded properties is violated. The BOOP toolkit could then be used to statically search for an execution path that violates the specified properties.

1.1 Related Work

Similar approaches are pursued by other research teams. We have already mentioned the SLAM project at Microsoft Research and we cover this approach in detail in this thesis.

The BLAST[5] toolkit of UC Berkeley [HJMS02] is based on the SLAM project, too, and uses "lazy abstraction". If the abstraction needs to be refined, only the parts that are affected by the new predicates are updated. This algorithm yields different predicate sets for different program points. Furthermore, unnecessary exploration of parts of the state space that are already known to be free of errors is prevented.

The *Java Path Finder*[6] (JPF) tool developed at NASA Ames Research Center is able to check invariants and search for deadlocks in Java bytecode programs [BHPV00]. The abstraction algorithm of JPF relies on user-specified abstraction criteria and uses the BANDERA[7] tool set [CDH+00] for static program slicing of concurrent Java programs. The advantage of JPF over our approach is that it is able to handle concurrent programs. Since JPF processes Java bytecode it is not necessary to have access to the source code of the libraries that the Java application uses. However, the necessity for user annotations makes the tool less acceptable for a non-scientific audience. Furthermore, the specification of abstractions for BANDERA by adding and removing variables to the abstract program requires a significant amount of user interaction.

Stefan Schwoon's model checker MOPED [ES01] is integrated into our toolkit and is discussed later in this thesis.

1.2 Organization of this Thesis

This thesis is organized as follows. Chapter 2 contains the preliminaries and notational conventions that are prerequisite for the following sections. Chapter 3 gives an overview of the approach by presenting the application of the BOOP tool on an example C program. We will refer to this example throughout the remaining sections. Chapter 4 explains the abstraction mechanism in detail. Chapter 5 shows how the C semantics is formalized in

[4] http://www.kernel.org
[5] `http://www-cad.eecs.berkeley.edu/~rupak/blast/`
[6] `http://ase.arc.nasa.gov/visser/jpf/`
[7] `http://www.cis.ksu.edu/santos/bandera/`

terms of the logic of the theorem prover HOL98. Chapter 6 gives a brief description of the principles that the model checking tool MOPED uses to determine the reachability of labels in Boolean Programs. Chapter 7 describes how the feasibility of paths in the C program is analyzed and how new predicates that are used to refine the abstraction can be discovered. Chapter 8 contains our conclusions, discusses open problems and summarizes uncompleted tasks and future work.

Chapter 2

Preliminaries

This chapter presents the concepts that are necessary to understand the matters presented in this thesis. This chapter covers propositional and higher order logic, Weakest Preconditions and Lattices. We give only a brief summary, since we assume that the reader is familiar with these basic concepts. For an in-depth treatment of these matters refer to the references we give in this chapter.

2.1 Formulæ and Models

This section contains a short description of the notation that we are using to represent formulæ and how such formulæ are interpreted in terms of models. This section should at least be skimmed in order to gain a unambiguous understanding of the notation that is used in the remaining chapters.

Throughout this thesis we make use of *propositional logic* and *higher order logic*, which are overlapping formalisms.

Definition 2.1. *(Syntax of propositional logic). A formula in propositional logic consists of* atomic formulæ A_i *(where* $i = 1, 2, 3, \ldots$*) and the logical connectives* $\wedge, \vee, \Longrightarrow$ *, and* $\neg$*. Formulæ are defined inductively:*

1. *All atomic formulæ are formulæ.*
2. $(\varphi_1 \wedge \varphi_2)$*,* $(\varphi_1 \vee \varphi_2)$*, and* $(\varphi_1 \Longrightarrow \varphi_2)$ *are formulæ if* φ_1 *and* φ_2 *are formulæ.*
3. *For every formula* φ*,* $\neg\varphi$ *is a formula.*

Later on we will use the notion of a *propositional combination* of predicates. A propositional combination of predicates is a propositional formula in which the atomic formulæ are arbitrary predicates.

These predicates may represent higher order logic terms. By *higher order logic* or *HOL logic* we denote the logic that is used by the theorem prover HOL98.

Definition 2.2. *The BNF grammar of terms in HOL logic is defined as follows:*

t_σ	$::=$	x_σ	*Variables*
	$\mid$	c_σ	*Constants*
	$\mid$	$(t_{\sigma' \to \sigma}\ t'_{\sigma'})_\sigma$	*Function applications (function* t*, argument* t'*)*
	$\mid$	$(\lambda x_{\sigma_1}.t_{\sigma_2})_{\sigma_1 \to \sigma_2}$	λ*-abstractions*

A λ-term $\lambda x.t$ denotes a function $v \mapsto t[v/x]$, where $t[v/x]$ denotes the result of substituting v for x in t. An application $t\ t'$ denotes the result of applying the function denoted by t to the value denoted by t'. Each term in the HOL logic is associated with a unique type. The notation t_σ is used to range over terms of type σ. (Examples for valid types would be $\mathbb{B}$ or $\mathbb{N}$.) A detailed definition of the HOL logic and its semantics can be found in [Uni01a]. In Chapter 5 we use HOL logic to formalize expressions of the C programming language.

Propositional formulæ are a subset of higher order logic terms. A definition of the logical connectives $\wedge, \vee$, and $\Longrightarrow$ in terms of higher order logic can be found in [Uni01a]. These connectives are in fact nothing but special cases of infixed HOL functions of type $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$. We will give a definition of their semantics in terms of *models*. A model $\mathcal{M}$ of a formula φ is an interpretation in which the formula becomes true. A model is therefore a tuple $(U_\mathcal{M}, I_\mathcal{M})$, where $U_\mathcal{M}$ denotes the domain of $\mathcal{M}$ and $I_\mathcal{M}$ is an interpretation function for all function symbols, variables and constants that occur in φ (if $\mathcal{M}$ models φ we write $\models_\mathcal{M} \varphi$). Furthermore, ψ is *valid* if *every possible* interpretation is a model of ψ. In this case we write $\models \psi$.

In general, each formula has potentially infinitely many models. There are always an unlimited number (a non-enumerable infinity) of models if there are any at all [Sch00, BBJ02]. Further on, we will denote the set (or class) of models of a formula ψ by $\mathcal{S}_\psi$.

A model $\mathcal{M}$ is a model of $\alpha \wedge \beta$ if and only if $\mathcal{M}$ models α as well as β. Similarly, $\models_\mathcal{M} (\alpha \vee \beta)$ if either $\models_\mathcal{M} \alpha$ or $\models_\mathcal{M} \beta$. If $\mathcal{M}$ models the negation of a formula φ then it cannot model the formula φ. Furthermore, a formula φ implies ψ iff every model of φ is also a model of ψ [End72]. Based on this, we can give following corollaries:

Corollary 2.3. *If $\mathcal{S}_\alpha$ is the set of models for α, and $\mathcal{S}_\beta$ is the set of models for β, then $\mathcal{S}_\alpha \cap \mathcal{S}_\beta$ is the set of models for $\alpha \wedge \beta$; similarly for $\vee$.*

Corollary 2.4. *A formula φ implies a formula ψ iff $\mathcal{S}_\varphi \subseteq \mathcal{S}_\psi$.*

Therefore, deciding if a formula φ implies a formula ψ is essentially a subset check. We will now consider conjunctions of the elements of a set Γ. Further on, we denote $\bigwedge_{\gamma_i \in \Gamma} \gamma_i$ as $\bigwedge \Gamma$.

Lemma 2.5. *If a conjunction $\bigwedge \Gamma$ implies a formula ψ, then ψ is implied by any conjunction $\bigwedge \Delta$ with $\Gamma \subseteq \Delta$.*

Proof. The conjunction of the elements of Δ is a larger conjunction than the conjuncted elements of Γ, since $\Gamma \subseteq \Delta$. Since $\wedge$ corresponds to an intersection of sets the models the set of models for Δ is given by $\mathcal{S}_\Gamma \cap \mathcal{S}_\Delta$. Then, $\mathcal{S}_\Gamma \cap \mathcal{S}_\Delta \subseteq \mathcal{S}_\Gamma$ implies that $\bigwedge \Delta$ implies $\bigwedge \Gamma$. Therefore the conjunction that corresponds to Δ implies ψ due to the transitivity of implication. □

Consider the intersection of the sets of models of a formula and its negation:

Lemma 2.6. $\not\models (\varphi \wedge \neg\varphi)$, *since false formulæ have no model; therefore* $\mathcal{S}_{(\varphi \wedge \neg\varphi)} = \emptyset$.

Therefore a false formula implies any arbitrary formula.

Lemma 2.7. *If $\bigwedge \Gamma$ implies $\neg\psi$, then there is no $\bigwedge \Delta$ with $\Gamma \subseteq \Delta$ that implies ψ (unless $\not\models \bigwedge \Gamma$ or $\not\models \bigwedge \Delta$).*

Proof. First note, that $\bigwedge \Gamma$ cannot imply ψ and $\neg\psi$ unless $\not\models \bigwedge \Gamma$. According to Lemma 2.5, every $\bigwedge \Delta$ must imply $\neg\psi$ if the conjunction of the elements of Γ implies $\neg\psi$ and $\Gamma \subseteq \Delta$. Therefore, the conjunction corresponding to Δ can only imply ψ if $\not\models \bigwedge \Delta$. □

2.2 Weakest Preconditions

In our abstraction algorithm we use formulæ (resp. *predicates*) to express assumptions about the state space of a program. Any assignment statement may have an impact on the truth values of these assumptions at the corresponding program point. The weakest precondition presented by Dijkstra [Dij76] is a formalism that allows us to capture the impacts of imperative statements on predicates. In [Gri81], Gries defines the Predicate Transformer $\mathcal{WP}(S, R)$ (or *weakest precondition*) to be the predicate that represents the set of *all* states such that execution of the statement S begun in any one of them is guaranteed to terminate in a finite amount of time in a state satisfying R:

Definition 2.8. *For a predicate φ and an assignment statement "`x := e`" where x is a scalar variable and e is an expression, the* weakest precondition of φ for the statement *"`x := e`" is defined by*

$$\mathcal{WP}(\text{``}\texttt{x := e}\text{''}, \varphi) = \varphi[e/x] \tag{2.1}$$

where $\varphi[e/x]$ denotes substituting all occurrences of x in φ by the expression e (this can alternatively be written as $(\lambda x.\varphi)e$, if lambda calculus is the preferred notation).

Example 2.1. *Let S be the assignment statement "`i := i + 1`" and let R be $i \leq 1$. Then*

$$\mathcal{WP}(\text{``}\texttt{i := i + 1}\text{''}, i \leq 1) = (i \leq 0)$$

for if $i \leq 0$, then the execution of "`i := i + 1`" terminates with $i \leq 1$, while if $i > 0$, the execution cannot make $i \leq 1$ [Gri81].

2.3 Lattices and Partially Ordered Sets

This section contains a brief description of lattices. A more detailed introduction to lattices can be found in [BvW98]. This section can be skipped by readers who are familiar with lattices.

Let A be a nonempty set and $\sqsubseteq$ a binary relation on A. If this relation is reflexive, transitive and antisymmetric we call $(A, \sqsubseteq)$ a *partially ordered set* (or *poset*). A partial order is total if $a \sqsubseteq b \vee a \sqsubseteq b$ holds for every $a, b \in A$. A partial ordered set A is bounded if it has a least element $\bot$ and a greatest element $\top$. For such posets $\bot \sqsubseteq a$ and $a \sqsubseteq \top$ for each $a \in A$.

Example 2.2. *The set of subsets $\mathcal{P}(A)$ and the set inclusion order $\subseteq$ form a partially ordered set. The least element is the empty set $\emptyset$ and the greatest element is the set A. $(\mathcal{P}(A), \subseteq)$ is total only if A is the emtpy set $\emptyset$ or a singleton set $\{a\}$.*

A function $\sqcap$ is called *meet* of a poset if it has following property:

$$(b \in B \implies \sqcap B \sqsubseteq b) \wedge ((\forall b \in B.a \sqsubseteq b) \implies a \sqsubseteq \sqcap B) \tag{2.2}$$

The dual function is $\sqcup$. It is called *join* of a poset and has following property:

$$(b \in B \implies b \sqsubseteq \sqcup B) \wedge ((\forall b \in B.b \sqsubseteq a) \implies \sqcup B \sqsubseteq a) \tag{2.3}$$

Example 2.3. *The partially ordered set $(\mathcal{P}(A), \subseteq)$ has the meet $\cap$ and the join $\cup$. Substituting $\cap$ and $\cup$ in Property 2.2 resp. Property 2.3 shows that the demanded properties hold:*

$$\begin{aligned} b_1 \cap b_2 \subseteq b_1 \text{ and } b_1 \cap b_2 \subseteq b_2 \\ a \subseteq b_1 \wedge a \subseteq b_2 \implies a \subseteq b_1 \cap b_2 \\ b_1 \subseteq b_1 \cup b_2 \text{ and } b_2 \subseteq b_1 \cup b_2 \\ b_1 \subseteq a \wedge b_2 \subseteq a \implies b_1 \cup b_2 \subseteq a \end{aligned}$$

A poset $(A, \sqsubseteq)$ is called a *lattice* if the meet $a \sqcap b$ and the join $a \sqcup b$ exists in A for all pairs a, b of elements of A.

Chapter 3

Overview

This chapter presents our abstraction/refinement algorithm by means of an example. We explain how our tool BOOP processes a program that loads a device driver module and accesses the corresponding device via the functions that the driver provides. (For a detailed description of dynamic kernel modules refer to [RC01].) We present the abstractions that BOOP generates and show how the tool automatically refines them until it finds a feasible execution path that is an example for an unintended behavior of the kernel module.

3.1 Architecture of BOOP

In Chapter 1 we have sketched the basic structure of the BOOP toolkit in terms of the phases that it (repeatedly) passes. In this section we will give an overview of the complete abstraction/refinement process and how it is implemented in our tool. The phases will be described in detail in the Chapters 4, 6, and 7.

The essential idea is to repeatedly apply an abstraction algorithm on a C program and check if the abstract model contains errors (with respect to a given specification). The abstract model is refined until it is detailed enough to decide if the original C program adheres to the specification or not. The upper half of Figure 3.1 shows a flow chart diagram that represents the control flow of the BOOP tool. (The bottom half shows implementation details that will be explained later in this chapter.)

In the flow chart diagram in Figure 3.1, the boxes with rounded edges represent the three phases (resp. modules) of our model checking tool. The boxes with fat borders represent the input and the output for the tool, and the remaining boxes denote the intermediate results.

A C program and a label that occurs in this program form the input for the BOOP toolkit. The first step is to generate a sparse model of the C program that reflects the control flow of the original C program. This first abstraction is usually very imprecise, since there is yet no information about which details of the C program have relevant influence on its control flow (i.e., the set of *predicates* that represent those details is empty). This model is then represented as *Boolean Program* (a detailed description of this formalism can be found in Chapter 4.2), which forms the input for the model checker.

The model checker searches for an execution path in the simplified program that leads to the specified label. Our abstraction algorithm ensures that the label is not reachable in the original C program if it is unreachable in the abstract model. (We denote this property as *soundness of the abstraction*. This matter is treated with more detail in [BMR02]). On

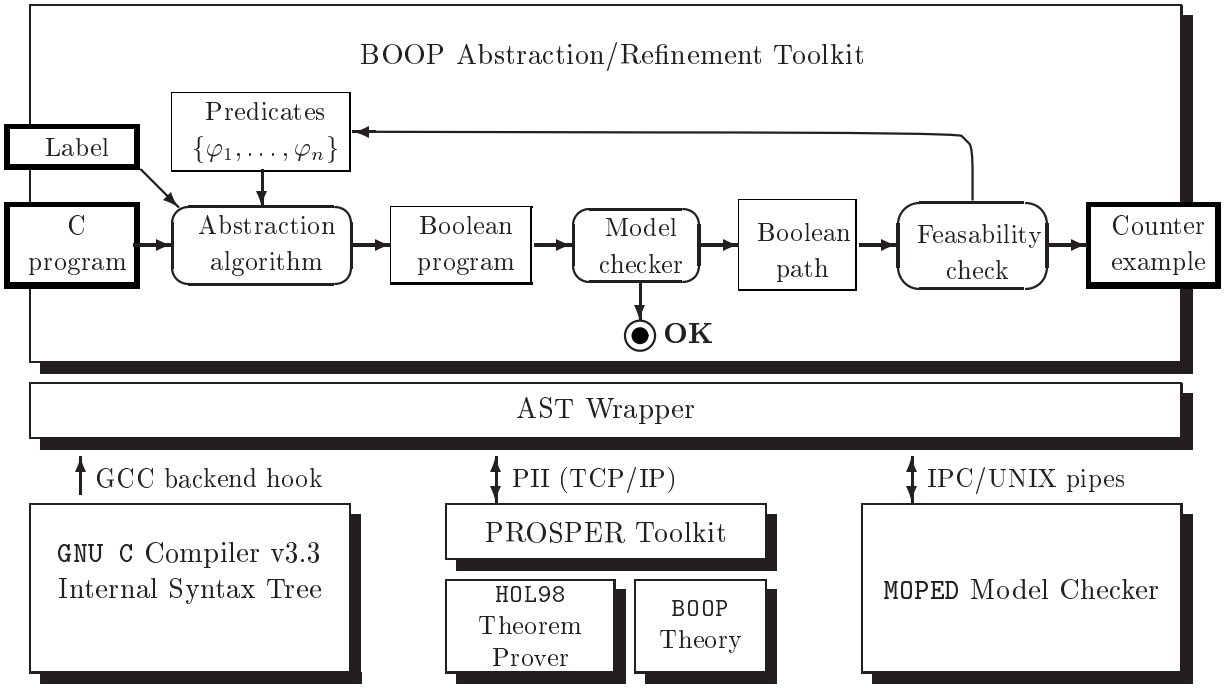


Figure 3.1: Overview of the BOOP Abstraction/Refinement Toolkit

the other hand, the existence of such a path does not per se imply an error in the original C program, because it might result from an insufficient model. Therefore, the path in the Boolean Program has to be mapped to a corresponding path in the original C program. In the case that BOOP can show the feasibility of this path in the original C program the erroneous trace is reported to the user as *counter example*.

If the error path is found to be infeasible, BOOP extracts a number of predicates from this spurious counter example that explain its infeasibility. These predicates represent details which are not reflected in the abstract model, but have relevant influence on the control flow of the program. Therefore, the tool starts over again and constructs a new abstract model that does not neglect these details.

There are certain programs for which the BOOP toolkit iterates the three phases over and over again without ever coming to a result. This is because the reachability problem is in general undecidable, since if there was a (complete) decision algorithm that could tell us if an arbitrary label ℓ is reachable in an arbitrary program $\mathcal{P}$ then we could use it to solve the halting problem. The user can therefore specify a maximum number of iterations.

The bottom half of Figure 3.1 shows the components that the BOOP toolkit is built of:

- We use the GNU C compiler to parse the C programs. The internal representation of the C program (which is described in detail in [Sta02]) is extracted via a backend hook and transformed to an abstract syntax tree (AST) that is used by the BOOP toolkit.
- The PROSPER toolkit [DCN+00][1] is used to integrate the HOL98 theorem prover into

[1] `http://www.dcs.gla.ac.uk/prosper/`

the BOOP toolkit. We supplied a HOL library that contains theorems and a decision procedure specific to our tool. Chapter 5 covers this library in detail.

- The model checker MOPED [ES01] is used to examine boolean programs for reachability of a specified label. Chapter 6 gives an introduction to the mechanism used by MOPED.

In the following section we will present how the BOOP toolkit works by means of an example.

3.2 A Linux Device Driver Example

A monolithic kernel architecture[2] places a number of restrictions upon systems which loadable modules were created to solve. Dynamically loadable modules enable the Linux Kernel to load device drivers on demand and to release them when they are not needed anymore. When a device driver is registered the kernel provides a major number that is used to uniquely identify the device driver. The corresponding device can be accessed through special files in the filesystem; they are conventionally located in the */dev* directory. If a process accesses a device file the kernel calls the corresponding `open`, `read`, and `write` functions of the device driver. Since a device driver must not be released by the kernel as long as it is used by at least one process the device driver must maintain a usage counter[3]. Figure 3.2 shows the (strongly simplified) `open` function of a device driver called `testdev`.

```
int testdev_open (struct inode *inode, struct file *filp)
{
   assert (MAJOR (inode->i_rdev) == major);
   MOD_INC_USE_COUNT;

   if (locked)
      return -ENODEV;
   locked = TRUE;
   return 0; /* success */
}
```

Figure 3.2: `open`-function for a device

The function `testdev_open` makes sure that the device can only be accessed by one process at a time. Of course a real device driver would provide a more sophisticated access control mechanism. Allowing only one process to access the device is a brute-force way and is best avoided [RC01]. For reasons of simplicity, we have also omitted the calls to `spin_lock` and `spin_unlock` that protect the global variable `locked` from concurrent access.

The Linux Kernel provides a number of macros to modify the usage count. The following macros are simplified but basically equivalent to these macros:

[2] A detailed disputation concerning monolithic kernels vs. micro kernels was posted by L.B. Torvalds and A.S. Tanenbaum in `comp.os.minix` in 1992 (`http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html`).

[3] In modern kernels, the system automatically tracks the usage count. However, device drivers that must be portable to older kernels will need to adjust the usage count manually.

```
    major = register_chrdev (0, "testdev", &testdev_ops);
    inode.i_rdev = major << MINORBITS;
    ...
    init_module ();
    ...
A:  testdev_open (&inode, &my_file);
    ...
    do
      {
B:      rval = testdev_open (&inode, &my_file);
        if (rval == 0)
          {
            count = testdev_read (&my_file, buffer, BUF_SIZE);
C:          testdev_release (&inode, &my_file);
          }
        else
          count = 0;
      }
    while (count != 0);
    ...
D:  testdev_release (&inode, &my_file);
    ...
    cleanup_module ();
    ...
E:  unregister_chrdev (major, "testdev");
```

Figure 3.3: Registering the device driver and accessing the device

```
#define MOD_INC_USE_COUNT  (usecount = usecount + 1)
#define MOD_DEC_USE_COUNT  (usecount = usecount - 1)
#define MOD_IN_USE         (usecount != 0)
```

The `open` function contains an obvious bug: The usage count is increased even if the device is locked and access cannot be granted. We will show how BOOP is able to find it and report an execution path that explains the inadequacy of the presented piece of code. Figure 3.3 shows a code snippet that loads the `testdev` device driver using the `register_chrdev` function, accesses it via the `testdev_open`, `testdev_read`, and `testdev_release` functions, and unloads it again.

A short look at the code in Figure 3.3 shows that the program attempts to open the device twice (at the locations $\mathcal{A}$ and $\mathcal{B}$) without closing it before. The second attempt is bound to fail, since the device driver maintains a lock that prevents simultaneous access to the device. At locations $\mathcal{C}$ and $\mathcal{D}$ the program makes calls to the `testdev_release` function (corresponding to the calls to `tesstdev_open` at $\mathcal{B}$ and $\mathcal{A}$). Finally, the program tries to unload the device driver at location $\mathcal{E}$.

We have replaced `register_chrdev` with a stub function that initializes the usage count to zero. `unregister_chrdev` examines the usage count and jumps to a label called `ERROR` if

(a) Abstraction 2 of testdev_open

```
bool<2> testdev_open
        ({inode→i_rdev≫ 8 =major})
begin
   decl {testdev_open = 0};
   assert ({inode→i_rdev≫ 8 =major});
   skip;
   if (*) then
      assume (!0);
      {testdev_open= 0} := choose (0,1);
      return {inode→i_rdev≫ 8 =major},
             {testdev_open= 0};
   else
      assume (!0);
   fi

   {testdev_open= 0} := choose (1,0);
   return {inode→i_rdev≫ 8 =major},
          {testdev_open= 0};
end
```

(b) Abstraction 3 of testdev_open

```
decl {locked≠ 0};
...
bool<2> testdev_open
        ({inode→i_rdev≫ 8 =major})
begin
   decl {testdev_open = 0};
   assert ({inode→i_rdev≫ 8 =major});
   skip;
   if (*) then
      assume ({locked≠ 0});
      {testdev_open= 0} := choose (0,1);
      return {inode→i_rdev≫ 8 =major},
             {testdev_open= 0};
   else
      assume (!{locked≠ 0});
   fi
   {locked≠ 0} := choose (1,0);
   {testdev_open= 0} := choose (1,0);
   return {inode→i_rdev≫ 8 =major},
          {testdev_open= 0};
end
```

Figure 3.4: 2nd and 3rd abstraction of `testdev_open`

`MOD_IN_USE` evaluates to true. We have thus provided a simple specification that expresses the constraint that the usage count must be zero when the program tries to release the device driver. The program adheres to this specification only if `ERROR` is not reachable.

In the first iteration, BOOP extracts all expressions that are parameters to a call to the function `assert` and constructs a Boolean Program with a very sparse control flow graph. In our example, the first predicate set E_0 contains only one predicate, namely {inode→i_rdev≫ 8 =major} which represents the expression `(MAJOR (inode->i_rdev) == major))`. Basically, the first abstract program allows *all* paths to be feasible. `ERROR` is reachable in this abstraction, because no restrictions are imposed on the control flow. MOPED reports the following execution trace: Choose the `then`-branch in `testdev_open` when this function is called at label $\mathcal{A}$ and $\mathcal{B}$, enter the `then`-branch of the conditional statement below $\mathcal{B}$, call `testdev_release` at label $\mathcal{C}$, and then continue the execution until `unregister_chrdev` is called at label $\mathcal{E}$. This execution trace leads to the label `ERROR`.

BOOP takes this execution trace, extracts the corresponding path from the original C program and checks if the path is feasible. The outcome is that the path that MOPED suggested is infeasible in the original C program, because label $\mathcal{C}$ can never be reached if the `then`-branch of `testdev_open` was executed at label $\mathcal{B}$. Now BOOP searches for predicates along

that path that explain its infeasibility. It finds out, that the return value of `testdev_open` influences the decision if `testdev_release` is called at label $\mathcal{C}$. Therefore, BOOP additionally considers the predicates {testdev_open= 0} and {rval= 0} for the construction of a more detailed abstract program. The former predicate adds information about the return value of `testdev_open` and the latter is used to keep track of the truth value of the guard of the `if`-statement below label $\mathcal{B}$.

```
   decl {rval= 0};
   ...
   register_chrdev ();
   skip;
   ...
   init_module ();
   ...
   prm1 := choose (0,0);
A: ret1,ret2 := testdev_open (prm1);
   ...
ℓ: skip;
   prm1 := choose (0,0);
B: ret1,ret2 := testdev_open (prm1);
   {rval= 0} := choose (ret2,!ret2);
   if (*) then
      assume ({rval= 0});
      testdev_read ();
C:    testdev_release ();
   else
      assume (!{rval= 0});
      skip;
   fi
   if (*) then
      assume (!0);
      goto ℓ;
   fi
   assume (!0)
D: testdev_release ();
   cleanup_module ();
E: unregister_chrdev ();
```

Figure 3.5: Abstraction of Figure 3.3

Figure 3.4(a) and Figure 3.5 present the abstract program that was automatically generated by BOOP during the 2nd iteration. The notation that is used to represent the abstract program is that of Boolean Programs. The syntax and semantics of Boolean Programs are discussed in Section 4.2 and defined in [BR00]. All we need to know about Boolean Programs for now is that they are essentially C programs in which the only allowed type is $\mathbb{B}$. Instances of this type are represented by **0** and **1**. Boolean programs allow non-determinism of the control flow, which is indicated by the conditional expression `*`.

The assign statements in the boolean program represent the effect of the C statement s at the corresponding position in the C program on the predicates in E_1. BOOP tries to express the boolean value of a predicate φ *after* the execution of the statement s in terms of **0**, **1** and the remaining predicates that are in scope at that point. However, it might be the case that the predicates do not provide sufficient detail to make an exact statement about the truth of φ. Boolean Programs provide a facility to express a non-deterministic assignment of truth values to predicates: the `choose` function. This function takes two parameters and returns **1** if the first parameter evaluates to **1** and **0** if the second parameter evaluates to **1**. If both parameters are **0** then `choose` non-deterministically returns either **0** or **1**. This means that φ can take either value and the model checker has to consider both cases.

The `testdev_open` function presented in Figure 3.4(a) sets the value of the return predicate {testdev_open= 0} to either **0** or **1**, depending on the chosen branch of the `if`-statement. The guard of an `if`-statement is always the conditional expression `*`. Restrictions on the control flow are imposed by a following call to the `assume` function. The model checker only considers paths that do not contain calls to `assume` with an actual parameter predicate that

	Calling function `register_chrdev`.	
	Assignment is executed.	!(locked≠ 0)
	Calling function `init_module`.	
$\mathcal{A}$:	Calling function `testdev_open`	
	Assertion holds.	
	Assignment is executed.	
	Condition is false; Jump over if-statement.	
	Assignment is executed.	
	Return from `testdev_open`.	(locked≠ 0)
$\mathcal{B}$:	Calling function `testdev_open`.	
	Assertion holds.	
	Assignment is executed.	
	Enter then-branch of conditional statement.	
	Return from `testdev_open`.	
	Enter else-branch of conditional statement.	
	Assignment is executed.	!(rval= 0)
	Exit do loop.	
$\mathcal{D}$	Calling function `testdev_release`.	
	Calling function `cleanup_module`.	
$\mathcal{E}$	Calling function `unregister_chrdev`.	
	Enter then-branch of conditional statement.	
`ERROR`:	Reaching label ERROR	

Figure 3.6: Explanation for violation of the specification

evaluates to **0**. In the second abstraction either branch of the `if`-statement can be chosen, since the fact that the open function maintains a lock is neglected.

Figure 3.5 shows the second abstraction of the code in Figure 3.3. The labels $\mathcal{A}$ to $\mathcal{E}$ relate the points in the code to the original program and ℓ is fresh label introduced by BOOP. The call to `testdev_open` at label $\mathcal{B}$ is of special interest: The second return value (which corresponds to the value of {testdev_open= 0} in Figure 3.4) provides the information that is necessary to decide if label $\mathcal{C}$ can be reached or not. We also see that in this abstraction it has obviously no relevance how often the `do`-loop is iterated.

BOOP calls MOPED for a second time to determine if the label `ERROR` (which is hidden in the call to `register_chrdev` at label $\mathcal{E}$) is reachable in the second abstract program. Again MOPED provides an example of how the label in question can be reached: The model checker suggests to select the `then`-branch of the `if`-statement in `testdev_open` each time that the function is called. Thus, the usage count is increased twice (in each call to `testdev_open`) but only decreased once (in the call to `testdev_release` at label $\mathcal{D}$). This is not far from the truth, but the path is still infeasible, since the `locked` variable in the original C program was initialized to `0` in `register_chrdev` and choosing `then`-branch in the first call to `testdev_open` is therefore no option.

BOOP determines that the corresponding path in the original C program is infeasible, because the value of the global variable `locked` was not considered. The tool discovers the predicate {locked≠ 0} that reflects the value of this variable, constructs a new predicate set

$E_2 = E_1 \cup \{\{\text{locked} \neq 0\}\}$ and calculates a new abstraction of the C program.

Figure 3.4(b) shows the third abstraction of the function `testdev_open`. Now the influence of the lock on the control flow is reflected. The call to `testdev_open` does only fail if the lock was already acquired. The abstract code in Figure 3.5 stays unmodified, since the new predicate has no influence on it.

Now MOPED is able to find an execution path that is feasible in the original C program. BOOP detects the feasibility and reports a detailed error path. We present a slightly simplified version of the output of BOOP in Figure 3.6. The detailed information about the locations of the statements is omitted; Furthermore we have suppressed most of the information that BOOP gives on the value of the predicates.

BOOP delivered a correct explanation for the inadequacy of the program. This counter-example provides the information the programmer needs to reproduce and correct the error. Deleting the line `MOD_INC_USE_COUNT;` and re-inserting it after the assignment `locked = TRUE;` would be a possible improvement of the program that makes `ERROR` unreachable. We cross-check by running BOOP on the modified program. After 4 iterations we yield the answer that `ERROR` is indeed not reachable anymore.

Chapter 4

Abstraction Algorithm

4.1 Introduction

In this chapter, we give a detailed description of the abstraction algorithm that is used to simplify the C program to such an extent, that it can be used as input to a model checking tool.

Abstraction, one of the key concepts of computer science, is the central technique for simplifying C programs to such an extent that state of the art model checking algorithms can be applied.

Abstracting away details from a model (which is in our case the investigated C program) yields a simpler model that still shares properties of interest with the original model, while superfluous information has been removed. To gain an appropriate abstraction algorithm for our problem, we have to define the properties we want to maintain and those we want to get rid of.

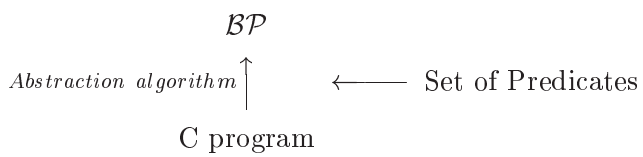


Figure 4.1: Generating a boolean program using Abstraction

Our intention is to test a C program for reachability of certain statements or program points. Reachability can be decided for programs with bounded state space (resp. finite state machines [Gor00]) and even for programs with an potentially unlimited state space if they contain only variables of Boolean type [EHRS00, ES01, BR00].

We need an abstraction mechanism that maps a C program to an abstract program that adheres to this restriction. This mechanism must preserve all possible execution traces of the C program. To obtain a *valid* abstraction we must take care that no traces are lost, otherwise we would forfeit completeness.

The execution traces (or paths) are determined by the truth values of the guard predicates of the conditional statements. Furthermore, the concrete states of the original C programs can be described by predicates [Gri81]. Predicate Abstraction is an approach where the concrete states of an infinite state system are mapped to abstract states according to their evaluation under a finite set of predicates [BMR02]. Each predicate can have the values *true* or *false*.

Furthermore, non-deterministic control flow is introduced whenever the predicates do not provide enough information about the concrete state to decide which branch of a conditional statement must be chosen. This approach does therefore fit our purposes.

At each program point, the state space of the abstraction is defined in terms of a *finite* set of predicates (refer to Chapter 7 for a description of how this set is obtained). The abstraction algorithm yields an overestimation of the concrete state space, because the potentially unlimited number of possible states can not be represented by a limited number of Boolean predicates. The impact of a C statement on the truth value of a predicate has to be expressed by using the limited information about the current state that the predicates supply. We use a theorem prover and Hoare Logic to calculate the *abstract state* transitions that approximate the changes of the *concrete state space* (the exact mechanism is explained in Sections 4.3 and 4.4). This introduces a certain inaccuracy with respect to the possible execution traces of the concrete program, resulting in superfluous paths and therefore potentially spurious counterexamples[1].

The abstraction can be presented using the control-flow constructs of the C programming language. However, all variables are restricted to be of Boolean type. We call such programs *Boolean Programs* ($\mathcal{BP}$) in accordance to the SLAM approach. Section 4.2 covers Boolean Programs in detail.

4.2 Boolean Programs

This section gives a description of syntax and semantics of Boolean Programs as they are presented in [BR00] and implemented in the MOPED model checker. An overview of the syntax is given in Figure 4.2.

4.2.1 Variables, Types, and Scopes

Boolean Programs are quite similar to C programs, apart from the fact that the type domain is restricted to $\mathbb{B}$. This means that Boolean Programs cannot contain integer variables, floating point variables, pointers or compound types (e.g. `structs`). Since there is only one type, variable declarations need not specify a type. The names identifying Boolean variables can be regular C-style identifiers or arbitrary strings enclosed in curled braces, e.g. {`a!=b`}. The latter form is a convenient way to associate *predicates* in another language such as C with Boolean variables. Boolean variables are either global (declared at the head of the Boolean program, outside the scope of a procedure) or local (declared inside the scope of a procedure). The scope of a local variable is restricted to the procedure where it is defined.

A constant in a Boolean Program can either be true or false. This is represented by **1** resp. **0**. Expressions can contain constants, variables, and logical connectives (see Figure 4.2). Expressions in Boolean Programs are always free of side effects.

4.2.2 Statements

This section covers the statements that do not influence the control flow of Boolean Programs.

1. **Skip Statement**. The `skip`-statement does not modify the state of the program. It is simply a placeholder for an instruction that has no effect. In fact, `skip`-statements

[1] Chapter 7 will explain how this case is handled

could be completely omitted without changing the semantics of the program. However, they simplify the task of associating statements in the abstract model (the Boolean Program) with with statements of the original model (the C program).

2. **Assignment Statement**. Boolean Programs may contain *parallel* assignment statements (as known from programming languages like OCCAM or CSP). A parallel assignment denotes the simultaneous assignment of the right hand side values to the corresponding left hand side values. This mechanism allows to exchange the values of two variables without using a helper variable, e.g. `a, b := b, a;`. This notation is very convenient for our approach, since a single assignment in our original model may change the value of more than one predicate in our abstract model.

4.2.3 Control Flow

At the beginning of this chapter we demanded that our abstract model reflects the control flow of our original C program. Boolean Programs contain a subset of the control-flow constructs of C and do therefore perfectly suit our purpose. The feasible traces are determined by the *deciders*. Deciders are either Boolean expressions which evaluate to **0** or **1**, or `*`, which evaluates to **0** or **1** non-deterministically. The non-determinism in Boolean Programs allows a single program to represent a multitude of feasible paths without the necessity for user interaction[2]. Whenever the abstraction algorithm is unable to provide enough detail to describe the exact behavior of the original model non-determinism will be used to handle this inaccuracy.

1. **Goto Statement**. The `goto`-statement unconditionally branches to a specified point in the program. It is equivalent to the `goto`-statement in C programs.

2. **Conditional Statement**. `if-then-else`-statements allow to pass over the control to one of two different program points (denoted *then*- and *else*-branch), depending on the value of the *guard* (a missing *else*-branch is equivalent to an *else*-branch that contains nothing else but a `skip`-statement). If the guard is a non-deterministic decider the model checker may choose either of the two branches. As we will see in Section 4.5.3 this will be the common case in our abstract model. In contrast to languages like OCCAM or PROMELA, conditional statements in Boolean Programs have only one guard. Therefore, the set of states in which the `if`-branch is chosen is the exact complement of the set of states in which the *then*-branch is chosen. In Section 4.5.3 we will see how this restriction can be weakened.

3. **Loop Statement**. Loops allow the (potentially unlimited) repetition of a sequence of statements. The syntax of `while`-loops is:

   ```
   while (decider) do
     sseq
   od
   ```

   The `while`-loop is the only kind of iteration statement in Boolean Programs.

[2] In fact, non-determinism is often used in model checkers to *simulate* user interaction, since programs should work correctly for any input

4.2.4 Procedures

Boolean Programs contain procedures with call-by-value parameter passing and recursion. Boolean procedures may return an arbitrary (but finite) number of values. The amount of storage a Boolean Program can access at any program point is finite (resp. bounded). However, since recursive calls to procedures are allowed the state space is potentially unbounded. Nevertheless, Boolean Programs are a class of programs for which reachability and termination is decidable. We will explain in Chapter 6 how this can be accomplished.

The `return`-statement returns the control to the calling procedure. Depending on the signature of the called procedure, it may return a number of values to the calling procedure.

Assertions form a special case of a procedure call. Assertions demand their actual parameters to evaluate to true. Any paths that lead to a violation of an assertion are reported by the model checker, since they lead to an abnormal termination of the program. An assertion statement can be understood as a call to the built-in `assert` procedure.

Assumptions are dual to assertions and are used to impose restrictions on the control flow. Paths that encounter a call to `assume` with a parameter that evaluates to **0** are infeasible and therefore ignored by the model checker.

Example 4.1. *Figure 3.5 shows an excerpt of a Boolean Program. At label $\mathcal{A}$ the parameter the parameter prm_1 is passed to the Boolean function* `testdev_open` *using call by value semantics. The function returns two Boolean values that are assigned to ret_1 and ret_2 in parallel.*

Figure 3.4 shows the definition of a Boolean function. The function returns a tuple of Boolean values. This is denoted by `bool<2>`*. The type of the formal parameter needs not be specified, since $\mathbb{B}$ is the only type in Boolean programs. The body of the function starts with the declaration of the predicate $\{testdev\_open = 0\}$. Again, the type is not specified. The declaration block is followed by a call to the function* `assert`*. At the end of the function, two Boolean values are returned.*

The `if`*-statements in Figure 3.5 show the use of non-deterministic deciders. The control flow is restricted by calls to the* `assume`*-function, which is the dual to* `assert`*.*

Finally, the statement `goto` *ℓ demonstrates the use of gotos.*

We have now completely defined Boolean Programs, which form the range of our abstraction function. The set of all valid C programs forms the domain. We assume that the reader is familiar with the C programming language. A detailed description of the C programming language can be found in the standard work of Kernighan and Ritchie [KR88].

We will now employ an algorithm that translates C programs to Boolean Programs with respect to a set of C predicates. The detailed description of this algorithm will be subject of the remaining part of this chapter.

Due to the restriction to a limited number of Boolean variables the behavior of the original C program can only be described with limited detail. These Boolean variables correspond to a set of arbitrary predicates over the unbounded state of a C program. Any C statement that modifies the state of the original C program (our *concrete* model) may cause a change of the value of the predicates at the corresponding program point.

Syntax			Description
prog	::=	$decl^*\ proc^*$	*A program is a list of global variable declarations followed by a list of procedure definitions*
decl	::=	**decl** id^+**;**	*Declaration of variables*
id	::=	$[\mathbf{a-zA-Z\_}][\mathbf{a-zA-Z0-9\_}]^*$	*An identifier can be a regular C-style identifier*
	|	$\{string\}$	*or a string of characters between '{' and '}'*
proc	::=	*id* (id^*) **begin** $decl^*$ *sseq* **end**	*Procedure definition*
sseq	::=	$lstmt^+$	*Sequence of statements*
lstmt	::=	*stmt*	
	|	*id* **:** *stmt*	*Labeled statement*
stmt	::=	**skip;**	
	|	**goto** *id*;	
	|	**return** $expr^*$**;**	
	|	id^+ **:=** $expr^+$**;**	*Parallel assignment*
	|	**if** (*decider*) **then** *sseq* **else** *sseq* **fi**	*Conditional statement*
	|	**while** (*decider*) **do** *sseq* **od**	*Iteration statement*
	|	**assert** (*decider*);	*Assert statement*
	|	*id* ($expr^*$);	*Procedure call*
decider	::=	*	*Nondeterministic choice*
	|	*expr*	
expr	::=	*expr binop expr*	
	|	!*expr*	*Negation*
	|	(*expr*)	
	|	*id*	
	|	*const*	
binop	::=	'$\vee$' | '$\wedge$' | '=' | '$\neq$' | '$\Longrightarrow$'	*Logical connectives*
const	::=	**0** | **1**	*False/True*

Figure 4.2: The syntax of Boolean Programs [BR00]

4.3 The Weakest Precondition of Assignment Statements

To capture the impacts of imperative statements on predicates, we will make use of the weakest precondition (see Definition 2.8 in Chapter 2) of assignment statements $S =$"`x := e`", where x denotes a scalar variable in a C program, and e is an *side effect free*[3] expression.

The computation of $\mathcal{WP}$ is an essential element of the abstraction algorithm, since its result yields information of how the predicates in the predicate set E are affected by an assignment statement. If $\psi = \mathcal{WP}($"`x := e`"$, \varphi)$ and $\psi \in E$ then the effect of "`x := e`" is that after the execution of this statement the predicate φ will be have the truth value that ψ had before its execution. However, in general it is not the case that ψ is contained in E.

Example 4.2. *Suppose that* $E = \{(x < 5), (x = 2)\}$. *The weakest precondition* $\mathcal{WP}($"`x := x + 1`"$, x < 5)$ *is* $x < 4$. *However,* $(x < 4) \notin E$, *and therefore we cannot replace the assignment statement "*`x := x + 1`*" by the new statement "*`{x < 5} := {x < 4}`*", since no predicate* `{x < 4}` *is defined in the abstraction. However, since* $(x = 2) \implies (x < 4)$ *we know that if the predicate* $(x = 2)$ *is true before the assignment statement, then* $(x < 5)$ *must be true afterwards [BMMR01].*

Obviously, in some cases it is possible to define the truth value of a predicate φ after the execution of an assignment statement in terms of a logical combination of the predicates in E. If this is not the case, we cannot predict which truth value the affected predicate φ will have after the execution and we will have to consider both the case that the predicate becomes *true* and the case that it becomes $false$. Therefore we replace the assignment by "φ`:= *`".

In Section 4.4 we will present an algorithm that tries to find a logical combination that implies the weakest precondition of the statement for a given predicate.

4.3.1 Handling Pointers and Aliasing

The formalism presented in [Gri81] (and the definition of the weakest precondition in Chapter 2) does not cover the semantics of pointers that are inherent in the C programming language. If the predicates or assignment statements contain pointers, then Equation 2.1 that was presented in Definition 2.8 is not necessarily correct. An assignment might change the truth value of a predicate even if none of the variables that the predicate directly mentions are directly mentioned in the assignment statement (by "directly mentioning" we mean a mention of the name of the variable):

Example 4.3. $\mathcal{WP}($"`x := 3`"$, *p > 5)$ *is not* $(*p > 5)$ *if* x *and* $*p$ *are aliases, since if this is the case,* $(*p > 5)$ *cannot be true after assigning* 5 *to* x *(resp.* $*p$*) [BMMR01].*

A similar problem occurs when the left hand side of the assignment is a pointer dereference. The result of ignoring these side effects is an abstract program that does not cover all the paths that are feasible in the original C program, if the latter contains pointers. This would imply that erroneous paths are not detected[4].

[3] An expressions that has side effects would potentially modify an additional variable. Such expressions are not covered by the definition given in Gries's book [Gri81]. They must be split up into several assignment statements, one for each variable that is modified

[4] This is the case at the current state of our implementation

This problem is addressed by the introducing the concept of *locations*[5]. A *location* may either be a variable or a dereference of a location[6]. We need to define an enhanced version of $\mathcal{WP}$ that considers not only variables but also locations.

Definition 4.1. *Let $\{y_1, y_2, \ldots, y_n\}$ be the set of locations that are mentioned in the predicate φ. Let x denote the location on the left hand side of the assignment statement. Then we define*

$$\varphi[x, e, y] = (\&x = \&y \wedge \varphi[e/y]) \vee (\&x \neq \&y \wedge \varphi) \tag{4.1}$$

(where & denotes the address operator of C) and

$$\mathcal{WP}(\text{“}\texttt{x := e}'', \varphi) = \varphi[x, e, y_1][x, e, y_2] \ldots [x, e, y_n] \tag{4.2}$$

in accordance to [BMMR01].

If the set of locations mentioned in φ that are aliased by x cannot be restricted by performing a pointer analysis [ASU86] then $\mathcal{WP}(S, R)$ will yield an expression of 2^n disjuncts, where each disjunct represents a possible aliasing scenario of the n locations with x. In the SLAM project, Manuvir Das's points-to algorithm [Das00] is used to estimate the set of aliases at each program point (see also [Ste96]). Usage of such an algorithm does improve the precision of the abstraction.

Our current implementation does not contain a pointer analysis algorithm.

4.4 Strengthening of Weakest Preconditions

In general, the weakest precondition $\psi := \mathcal{WP}(\text{“}\texttt{x := e}\text{”}, \varphi)$ of an assignment instruction has no logical equivalent in the set E of predicates that are in scope at the corresponding program point. Therefore, the value of the predicate φ after execution of “`x := e`” must be approximated by means of a logical combination of the predicates $\varphi_i \in E$. In this section, we present an algorithm that calculates the *weakest* combination of predicates that implies the weakest precondition ψ. In accordance to [BMMR01], we will call this procedure “*strengthening*[7] a precondition ψ to an expression over the predicates in E” and denote the corresponding function by $\mathcal{F}(\psi, E)$. Additionally, a function $\mathcal{G}(\vartheta, E)$ that calculates the *strongest* combination of predicates that is implied by an expression ϑ will be defined as counterpart to $\mathcal{F}$. The procedure of calculating $\mathcal{G}(\vartheta, E)$ will be called “*weakening* ϑ”.

4.4.1 Calculating $\mathcal{C}(\psi, E)$

As a prerequisite for the strengthening algorithm, we define a function $\mathcal{C}(\psi, E)$ that calculates the *weakest* conjunctions of predicates $\varphi_i \in E$ and their negations $\neg\varphi_i$ that imply ψ. Such conjunctions will further on be called *cubes*.

The algorithm that we use to calculate $\mathcal{C}(\psi, E)$ involves the following steps:

1. Construct an augmented set E' that contains all predicates $\varphi_i \in E$ and their negations $\neg\varphi_i$. Furthermore, start with an empty *work queue* and an empty *result set*.

[5] The approach we present here is the one used in the SLAM project. It is based on Morris's Treatment of pointers [Bor00]

[6] Note that we pursue a different approach for accessing structure fields than used in the SLAM project. A detailed description of our approach will be given in Chapter 5.

[7] This is because the algorithm calculates the *weakest* compound proposition that is *stronger* than ψ

2. Assign an arbitrary (but total) order to the predicates $\varphi_i \in E'$.

3. Insert the empty set $\emptyset$ into the *work queue*.

4. Remove the front element Γ from the *work queue*. Check if there exists a set Δ in the *result set* such that $\Delta \subseteq \Gamma$. If such an element exists, discard Γ and continue with step 6. (We could also consider all sets with a cardinality of one in the work queue, negate their elements and check if (at least) one of the thus gained corresponding singleton sets is a subset of Γ. If this is the case, Γ can also be discarded.)

5. Construct the conjunction
$$\phi := \bigwedge_{\gamma_i \in \Gamma} \gamma_i \tag{4.3}$$
and let the theorem prover decide if either $\phi \implies \psi$ or $\phi \implies \neg\psi$ is a tautology (note that ϕ is *true* if $\Gamma = \emptyset$). If one of these propositions can be proven to be true, tag Γ with $\top$ or $\bot$ (depending on the result of the decision procedure) and insert it into the *result list*. Otherwise, find the greatest element $\varphi_k \in \Gamma$ (with respect to the order introduced in step 2), construct the sets
$$\Gamma \cup \{\varphi_i\} \text{ for each } i \text{ with } k < i \leq |E'| \text{ and } \neg\varphi_i \notin \Gamma \tag{4.4}$$
and append them at the end of the *work queue*. Discard Γ.

6. If the *work queue* is empty, terminate the algorithm and return the *result set*. Otherwise continue with step 4.

4.4.2 Calculating $\mathcal{F}(\psi, E)$ and $\mathcal{G}(\psi, E)$

Based on $\mathcal{C}$, the result of *strengthening* an expression ψ with respect to a predicate set E can easily be defined to be

$$\mathcal{F}(\psi, E) := \bigvee_{\Gamma_{\top,i} \in \mathcal{C}(\psi,E)} \left(\bigwedge_{\varphi_j \in \Gamma_{\top,i}} \varphi_j \right) \tag{4.5}$$

$$\mathcal{F}(\neg\psi, E) := \bigvee_{\Gamma_{\bot,i} \in \mathcal{C}(\psi,E)} \left(\bigwedge_{\varphi_j \in \Gamma_{\bot,i}} \varphi_j \right) \tag{4.6}$$

$$\tag{4.7}$$

Note that $\mathcal{F}(\psi, E)$ is false if $\mathcal{C}(\psi, E) = \emptyset$. $\mathcal{G}(\psi, E)$ can now be defined in terms of $\mathcal{F}$:

$$\mathcal{G}(\psi, E) := \neg\mathcal{F}(\neg\psi, E) \tag{4.8}$$

4.4.3 Efficiency Analysis

In this section, we will consider the space and time complexity of the algorithm that we use to calculate $\mathcal{F}(\psi, E)$.

- **Space Complexity**. Each set E of cardinality n has exponentially many subsets (including the empty set and E itself). Thus, the cardinality of the power set of E would

be 2^n, but we gain 3^n possible subsets if we also take the negated elements of E into consideration. The algorithm that calculates $\mathcal{C}(\psi, E)$ as presented above enumerates these subsets in ascending order with respect to their cardinalities. It prevents that all 3^n subsets have to be held in memory at one time, however, in the worst case (when the theorem prover cannot decide any of the propositions), we will still have to consider all of them. In the s^{th} step, a maximum of $2^s \cdot \frac{n!}{s! \cdot (n-s)!}$ subsets will be in memory, what is still $O(2^n)$ and therefore no significant improvement over the naïve approach. Below we will present an optimization of the algorithm that would yield a space complexity of $O(n)$ and we will explain, why we did not use this approach for our implementation.

- **Time Complexity**. As mentioned above, if the worst case occurs, we are forced to consider all 3^n possible subsets of E. As long as we are not willing to sacrifice some precision the number of subsets that we have to inspect cannot be reduced. Therefore, the time complexity of our algorithm is $O(3^n)$.

Since the running time of the algorithm is dominated by the cost of theorem proving (due to the exponential number of calls to the theorem prover), we found out that if the worst case occurs for large predicate sets, the algorithm takes unacceptably long time to terminate. In industrial projects, where the software is usually compiled at least once a day, users are unlikely to accept compile times of several hours or even days. Therefore, since $\mathcal{F}(\psi, E)$ is calculated at least once for every program point, we have to rely on that

- the set of predicates can be cut down by a syntactic cone-of-influence computation before the procedure that calculates $\mathcal{F}$ is called
- the propositions that occur in the earlier steps of the algorithm can be decided, thus yielding a reduction of the number of sets we have to consider.

As mentioned above, the *space complexity* of the algorithm could still be optimized. The subsets of E' (with $\mid E' \mid = n$) are isomorphic to n-bit integers. A bit that is set indicates that the corresponding predicate is contained in the subset that is represented by a n-bit integer. An algorithm that enumerates n-bit integers in an appropriate order could thus be used to construct an algorithm that does not store more than one subset at a time in memory.

For reasons of simplicity, we did not implement this algorithm. Even the usage of this more complicated algorithm that constructs only one subset at a time would make exponentially many calls to the theorem prover in the worst case. Thus we found the algorithm presented above to be sufficient, since in the worst case, the user would not be willing to wait for the termination of the algorithm anyway. Therefore, we decided that it does not matter if the algorithm runs out of memory sooner or later.

In this context it is interesting to note that if the above mentioned worst case occurs, the resulting line in the abstraction is an assignment of `choose (0, 0)` (what is equivalent to non-deterministic assignment) to the affected predicate.

4.4.4 Proof of Correctness

In this section we will show that the algorithm presented above does indeed yield the *weakest* propositional combination[8] consisting of predicates φ_i and the operators $\neg, \wedge$ and $\vee$ that

[8]The predicates φ_i may of course represent non-propositional expressions. However, by *propositional combination* we denote a formula consisting of the predicates φ_i and the mentioned operators that is constructed as if the φ_i were atomic formulæ, see Chapter 2

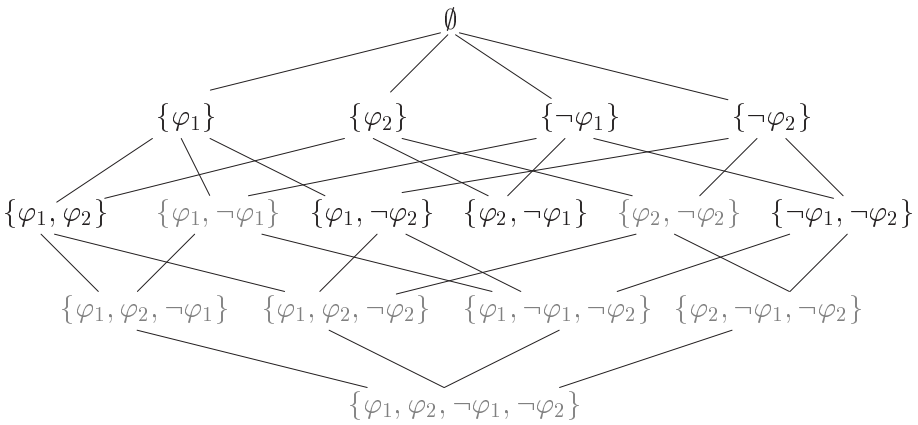


Figure 4.3: Lattice of predicates for $|E| = n$

implies ψ. For this purpose, we will need a rigorous definition of what we mean by "weak", which we will give in terms of models.

Intuitively, an assertion or weakest precondition ψ restricts the set of valid states at a program point. Thus, the set $\mathcal{S}_\psi$ of all models of ψ corresponds to the set of valid states at the affected point in the program. Furthermore, each predicate $\varphi_i \in E$ represents a set $\mathcal{S}_{\varphi_i}$ of models for φ_i.

The *weakest* logical combination $\varphi = f(E)$ of predicates that implies ψ can therefore be understood as the largest set of models $\mathcal{S}_{f(E)} \subseteq \mathcal{S}_\psi$ that can be represented by a formula consisting of the predicates in E and the connective symbols $\neg$, $\wedge$ and $\vee$.

Theorem 4.2. *The algorithm for calculating $\mathcal{F}(\psi, E)$ presented in Section 4.4 calculates the* largest *subset $\mathcal{S}_{\mathcal{F}(\psi,E)} \subseteq \mathcal{S}_\psi$ that can be represented by a propositional combination of the predicates in E.*

Proof. Since for all propositional formulæ there is an equivalent formula in Disjunctive Normal Form (DNF) [Sch00], it suffices to consider all possible disjunctions of conjunctions of predicates $\varphi_i \in E$ to cover all logical combinations. For reasons of readability, we will further on allow to interpret a set Γ of predicates (i.e., a *cube*) as the corresponding conjunction of these predicates:

$$\Gamma \equiv \bigwedge_{\gamma_i \in \Gamma} \gamma_i \tag{4.9}$$

Whenever it is not clear from the context whether we mean the formula represented by this conjunction or the *set* of predicates we will explicitely mention to which interpretation we refer.

In step 1 of our algorithm that calculates $\mathcal{C}(\psi, E)$ an augmented set E' that contains all predicates $\varphi_i \in E$ and their negations is constructed. Then, the set $\mathcal{P}(E')$ of all subsets of E' forms a complete lattice under the set inclusion ordering [CGP99, BvW98]. Figure 4.3 shows such a lattice for $E = \{\varphi_1, \varphi_2\}$ with $n = |E|$. The least element of the lattice is the empty set $\emptyset$, whereas the greatest element is the set E'. The meet of the lattice is $\cap$ and $\cup$ is the join.

If we construct the conjunction of the predicates in each subset that is element of the above mentioned lattice, then we get all possible conjunctions (or *cubes*) of predicates in E.

The subsets that would yield a trivial falsity (because they contain $\neg\varphi_i$ and φ_i) are depicted gray in Figure 4.3. Therefore, only 3^n of the 2^{2n} subsets are of interest.

Let us consider the sets of models that correspond with the above mentioned conjunctions. Let $\mathcal{S}_{\mathcal{P}(E')}$ be the set of sets of models that correspond to the subsets in $\mathcal{P}(E')$. Obviously, $\mathcal{S}_{\mathcal{P}(E')}$ does also form a lattice under the set inclusion ordering (however, note that the lattices $\mathcal{P}(E')$ and $\mathcal{S}_{\mathcal{P}(E')}$ are not homomorphic, i.e. the function that assigns sets of models to formulæ is not a lattice homomorphism). $\cap$ is the meet ($\sqcap$) of the lattice and corresponds to the logical connective $\wedge$, whereas $\cup$ is the join ($\sqcup$) (corresponding to $\vee$). The least element ($\mathcal{S}_\bot$) is the empty set $\emptyset$, corresponding to the formula $false$. The greatest element is the set $\mathcal{S}_\top$ of all models that model the formula $true$.

In step 4, the algorithm that calculates $\mathcal{C}(\psi, E)$ traverses the lattice $\mathcal{P}(E')$ in *Breadth First Search* (BFS) order[9], thus considering the elements in ascending order with respect to their cardinalities. All subsets representing trivial falseties are ignored, since the corresponding set of models would be empty due to Lemma 2.6. (We want to find the largest subset of $\mathcal{S}_\psi$, and any non-empty set is larger than $\emptyset$). Furthermore, if the result set contains a subset Δ of the investigated element Γ, we can safely discard Γ, since $S_\Gamma \sqcap S_\Delta \sqsubseteq S_\Gamma$ (due to the Lemmas 2.5 and 2.7). The order of traversal we chose guarantees that neither Γ nor one of its supersets would contribute a set of models for ψ that is larger than what we have already found. (Each superset Δ of Γ will imply ψ if $\Gamma \in \mathcal{P}(E')$ implies ψ.)

In step 5 we check if $\mathcal{S}_\Gamma$ with is a subset of $\mathcal{S}_\psi$ or of $\mathcal{S}_{\neg\psi}$. If so, we can tag it and add it to the result set.

We have thus shown that the algorithm that we use to calculate $\mathcal{C}(\psi, E)$ yields the largest sets of models that can be represented by conjunctions, since we have considered all cubes that can be constructed from the predicates in E.

The sets of models that the algorithm yields are in general disjoint. We gain the distributive union of these sets by constructing the disjunction of the conjunctions that correspond to the elements of the lattice $\mathcal{P}(E')$ that are result of $\mathcal{C}(\psi, E)$ (since the join function of the lattice $\mathcal{S}_{\mathcal{P}(E')}$ corresponds to $\vee$).

Assume that there is a formula F^+ consisting of the predicates in E and the logical connectives $\neg, \wedge$ and $\vee$ that has the same set of models as the formula F that our algorithm yielded plus at least one additional model $\mathcal{M}^+$. Then this formula F^+ can be represented in Disjunctive Normal Form, and the model $\mathcal{M}^+$ must be element of (at least) one of the sets S_{ϕ_i} that can be represented by conjunctions of the predicates in E and their negations. The formula F^+ can only contain conjunctions that correspond to the elements of the lattice $\mathcal{P}(E')$. This would imply that there is an element $\Delta \in \mathcal{P}(E')$ that was not found by the algorithm we used to calculate $\mathcal{C}(\psi, E)$. But this would mean that this algorithm has either found an element $\Gamma \subset \Delta$ or that the conjunction that corresponds to Δ does not imply ψ. Therefore, no such formula F^+ can exist. □

Note that the proof presented in Section 4.4.4 is only valid under the assumption that the decision procedure of the theorem prover we use is strong enough to decide all propositions that are constructed in step 5 of the algorithm we presented in Section 4.4.1. We will explain in Section 5.7 why such a decision procedure cannot exist. For this reason, the result that our algorithm yields can only be an approximation of the result that we expect. However, since we assume that the decision procedure is correct, the implementation of the algorithm that calculates $\mathcal{F}(\psi, E)$ cannot yield a formula that does *not* imply ψ.

[9] At least in the worst case, when none of the branches of the tree that represents the lattice can be pruned.

4.4.5 Optimizations

In Section 4.4.3 we mentioned that the runtime of the algorithm for $\mathcal{F}(\psi, E)$ is exponential in $|E|$, what means that the algorithm becomes infeasible for large sets of predicates. Therefore, we try to cut down the set of predicates before calling the procedure that calculates $\mathcal{F}(\psi, E)$. For this purpose, we make use of the fact that when we want to know whether or not $\mathcal{M}$ is a model for a formula φ, we don't need all of the (infinite amount of) information the interpretation gives us. All that matters are the values of the (finitely many) variables which occur free in φ [End72]. This means that if the set of free variables of a formula φ does not intersect with the set of free variables of a second formula ψ, then φ cannot imply ψ (due to the semantics of implication that was defined in Chapter 2). We can therefore restrict the set of predicates that are candidates for being part of a cube that implies the weakest precondition by performing a *syntactic cone-of-influence* computation [BMMR01].

Let $vars(\phi_i)$ denote the function that yields all free variables of the formula ϕ_i. Then the *syntactic cone-of-influence* computation corresponds to the *least fixpoint* [CGP99]

$$\mu Z.\{\psi\} \cup \{\varphi_i \mid vars(\varphi_i) \cap \left(\bigcup_{\phi_j \in Z} vars(\phi_j) \right) \neq \emptyset \wedge \varphi_i \in E\} \tag{4.10}$$

The computation starts with the empty set $E^- := \emptyset$ of predicates, and successively adds the predicates from E to E^- that have a set of free variables that intersects with one of the sets of free variables from either ψ or the elements of the actual set E^-. The computation stops if it reaches an iteration that adds no more elements to E^-.

This computation is also correct in the presence of dereferences or aliases of the locations to which the free variables of the expression correspond, since the weakest precondition mentions the dereferenced locations explicitely (according to Definition 4.1). However, our current implementation cannot handle aliases because of the lack of a pointer analysis.

Furthermore, we abstain from making a call to the theorem prover, when the assignment statement has no effect on the predicate (i.e. $\lambda\varphi.\mathcal{WP}(\text{"}\texttt{x := e}\text{"}, \varphi)$ is the identity function for the predicate), since in this case, the truth value of the predicate is not affected by the side effects of the assignment statement.

In the SLAM project several additional optimizations were applied [BMMR01]:

- Several syntactic heuristics are tried to construct $\mathcal{F}(\psi, E)$ directly from ψ (these include a check if $\psi \in E$).
- All computations by the theorem prover (as well as the results of the alias analysis) are cached, so that work is not repeated.
- As an approach that trades precision for efficiency, the length of cubes considered by $\mathcal{F}(\psi, E)$ is restricted to a constant k.

These optimizations were not implemented as part of this thesis and are left for future projects.

4.5 Abstraction of C constructs

This section provides a detailed description of how the predicate abstraction of various C constructs is calculated. All C constructs that our current implementation covers are de-

scribed. It will be explicitely mentioned whenever our implementation does not keep step with the presented specifications. This section does also contain suggestions for improvement of our implementation and hints of how various problems could be bypassed. Most of the abstraction algorithms that are presented in this chapter are taken from [BMMR01].

4.5.1 Restrictions

The BOOP toolkit supports only a subset of ANSI C. Most of the restrictions are of artificial nature and result from the limited amount of resources that we were able to invest into this thesis. We give a short summary of these restrictions:

1. **Expressions with side effects**. When the ANSI C standard talks about expressions, it usually means *expressions with side effects*. For reasons that we will explain in Section 5.2.1, there is no equivalent for such expressions in the logic of the theorem prover that we use. At the moment, expressions with side effects are not supported by our implementation. Therefore, we assume that the C program has been converted into a simple intermediate form in which all expressions are free of side effects. It is no great problem to automatically rewrite a C program expressions with side effects to a new C program that adheres to this restriction.

2. **Pointers and aliases**. For the reasons that were mentioned in Section 4.3.1 pointers and aliases are not yet handled by our implementation.

3. **Undefined Functions**. At the moment, our implementation cannot handle undefined or external functions without a function body, or functions with a variable number of arguments. This is a severe restriction for "real world" C programs, because the user would have to write stubs for all functions that he imported from external libraries. It is therefore an urgent task to extend our implementation so that it can handle this case. This is a non-trivial problem, since such a function can potentially modify the whole state-space, i.e. we would have to assume that the values of all global predicates are unknown after a call to an external function.

4. **Scopes**. ANSI C allows to introduce a new scope at arbitrary program points by insertion of a new code block. This facility is not reflected in Boolean Programs (see Section 4.2), where the only places where variables (resp. predicates) can be declared are the beginning of the program or at the beginning of a body of a function. This means that a variable of nested scope in a C program would have to be translated to either global scope or to the local scope of the procedure that encloses the variable. This task can be accomplished by variable renaming. Our implementation does not provide this functionality, therefore, it cannot handle C programs that contain nested scopes.

5. **Unsupported control constructs**. There are several C control flow constructs that our current implementation cannot handle. These include `switch` constructs and `for`-loops, and the statements `goto`, `break`, and `continue`. Their translation would be straight forward. However, they are not yet supported by our implementation.

6. **GNU extensions**. Though our implementation is based on the GNU compiler collection, none of the GNU extensions to the ANSI C standard (like inline assembler, complex numbers, statement-expressions, etc.) are supported.

Note that our implementation will abort if the input C program contains elements that we cannot translate yet.

4.5.2 Assignment Statements

Our abstraction algorithm translates assignment statements of C programs to parallel assignment statements. A C assignment statement potentially changes the values of all predicates (resp. Boolean variables) in scope. Assume that "`x = e;`" is an assignment statement at location ℓ in our C program. By applying the predicate transformer $\mathcal{WP}$ on every predicate that is in scope at the program point ℓ in the Boolean Program we compute the predicates that are possibly affected by the assignment statement. A predicate ψ can have the value **1** after the execution of the assignment statement at ℓ if $\mathcal{F}(\mathcal{WP}(\text{“}\texttt{x = e}\text{”}, \psi), E)$ holds before ℓ. Correspondingly, the predicate will have the value **0** if the strengthened weakest precondition of $\neg\psi$ holds before ℓ. Unless result of $\mathcal{F}$ is false, it can only imply either ψ or $\neg\psi$. If no predicate combination that is not false and that implies either ψ or $\neg\psi$ can be found, we assign the non-deterministic value `*` to the predicate.

Depending on the results of $\mathcal{F}(\mathcal{WP}(\text{“}\texttt{x = e}\text{”}, \psi), E)$ and $\mathcal{F}(\mathcal{WP}(\text{“}\texttt{x = e}\text{”}, \neg\psi), E)$ we have to decide which value we assign to the predicate ψ at the location ℓ. We use a procedure `choose` to calculate this value. This function is defined as follows:

```
bool choose (pos, neg)
begin
  if (pos) then
    return 1;
  else
    if (neg) then
      return 0;
    else
      return unknown ();
    fi
  fi
end
```

It makes use of the procedure `unknown` that non-deterministically returns either **1** or **0**. This function is defined as follows:

```
bool unknown ()
begin
  if (*) then
    return 1;
  else
    return 0;
  fi
end
```

The parallel assignment for the set of predicates $\{\psi_1, \ldots, \psi_n\}$ that are in scope at ℓ is

ℓ:
$\psi_1, \ldots, \psi_n$:=
  choose ($\mathcal{F}(\mathcal{WP}(\text{“x = e”}, \psi_1), E)$, $\mathcal{F}(\mathcal{WP}(\text{“x = e”}, \neg\psi_1), E)$),
  ...,
  choose ($\mathcal{F}(\mathcal{WP}(\text{“x = e”}, \psi_n), E)$, $\mathcal{F}(\mathcal{WP}(\text{“x = e”}, \neg\psi_n), E)$);

Example 4.4. *The statement* `a, b := b, a;` *exchanges the values of* `a` *and* `b` *without necessity of an additional variable.*

4.5.3 Gotos and Conditionals

Every `goto` statement in the C program can simply be copied to the Boolean Program. The label is not changed, since existing labels are generally taken over from the C program. However, `goto`s are not yet supported by our implementation.

The translation of conditionals is a bit more tricky, since a branch of a conditional must not be executed if the corresponding guard γ does not evaluate to true. As for the weakest preconditions of assignment statements, it is in general not the case that the predicate set we use for our abstraction contains a predicate that is equivalent to the guard γ of the conditional statement. We must therefore try to approximate the predicate γ by means of the predicates that are in scope at the corresponding program point in the Boolean Program. An underestimation of the states in which γ evaluates to true must be avoided – it would lead to a Boolean Program that is no valid abstraction (for the definition of a *valid* abstraction refer to the beginning of Chapter 4) of the C program that is inspected. That means that we must replace γ by a formula that is *weaker* or equal. This can be accomplished by replacing it with $\mathcal{G}(\gamma, E)$, i.e. by weakening γ over the predicates in E (refer to Section 4.4 for the definition of $\mathcal{G}(\gamma, E)$). If $\mathcal{G}(\gamma, E)$ holds, then the *then*-branch of the conditional may be executed. Correspondingly, if $\mathcal{G}(\neg\gamma, E)$ holds, then the *else* branch of the conditional statement may be executed. Note that the branches of a conditional statement in a Boolean Program are not mutually exclusive; that means that in some states it may be possible to execute either of the branches. This agrees with the idea of overestimating the traces that are feasible in the original C program. Spurious traces are sorted out in a later phase of the abstraction/refinement algorithm.

Now consider a C program that contains a conditional statement with the guard γ. Assume that the first statement in the *then*-branch is labeled with ℓ_1 and the first statement in the *else*-branch is labeled with ℓ_2 (this is in general not the case, but it makes it easier to tell the branches apart). The conditional statement is then translated as follows:

C program

if (γ) {
ℓ_1: ...
} else {
ℓ_2: ...
}

Boolean program

if (*) then
  assume ($\mathcal{G}(\gamma, E)$);
ℓ_1: ...
else
  assume ($\mathcal{G}(\neg\gamma, E)$);
ℓ_2: ...
fi

In the Boolean Program on the right side, E denotes the set of predicates that is in scope at the program point where the conditional statement occurs. The `assume` function that is called at the beginning of each branch assures that the branch can only be entered if its parameter evaluates to true (it is therefore the dual of `assert`). Traces in which `assume` is called with a parameter that evaluates to false are ignored by the model checker.

Example 4.5. *Figure 3.5 shows the translation of conditional statements with help of non-deterministic deciders. The* then*-branch of the conditional statement below label $\mathcal{B}$ is only feasible if $\{rval = 0\}$ evaluates to* ***1*** *when the branch is entered. Conversely, the* else*-branch can only be entered when $\{rval = 0\}$ evaluates to* ***0*** *at its entry point.*

In this example, the guard for the else*-branch is the negation of the guard of the* then*-branch. However, the sets of states in which the two guards evaluate to* ***1*** *may also overlap, allowing the execution of either branch. The branch will be chosen non-deterministically then, i.e. the model-checker has to consider both paths.*

4.5.4 Loops

By explaining how `goto`-statements and conditional statements are handled we have already provided the fundamentals for translating loops. Loop constructs can always be simulated by means of `goto`s and `if-then-else` constructs. In our current implementation it is still necessary to replace `for`-loops this way, since only `while`- and `do-while` loops are supported at the moment.

1. **While Loops**. `while`-loops can be translated relatively straight forward, because a similar construct is provided in Boolean Programs. Therefore, all we have to do is to replace the loop guard γ with `*` and prepend the statement `assume` $(\mathcal{G}(\gamma, E))$`;` (where E denotes the predicates in scope at the affected program point) to the loop body.

2. **Do-While Loops**. `do-while`-loop constructs have to be replaced by an equivalent `if (...) then goto` construct. For this purpose, we have to introduce a fresh label ℓ_{dw} and insert it before the body of the loop. When all statements of the body have been executed the `while`-construct of the loop is simulated by a conditional statement.

C program	**Boolean program**
`do {`	ℓ_{dw}:
ℓ_1: ...	ℓ_1: ...
`} while (`γ`);`	`if (`$\mathcal{G}(\gamma, E)$`) then`
	`  goto` ℓ_{dw}`;`
	`fi`

   As usual, γ and E denote the guard resp. the set of predicates in scope. ℓ_1 is provided to make clear where the loop body must be inserted.

3. **For Loops**. `for`-loops are not supported at the moment, since they are usually used in connection with expressions with side effects. According to K&R [KR88] the `for`-statement

```
for (expr1; expr2; expr3)
   statement
```

is equivalent to the `while`-loop

```
expr1;
while (expr2) {
   statement
   expr3;
}
```

Obviously both constructs rely heavily on the side effects of $expr_1$ and $expr_3$. Since our implementation does not yet support expressions with side effects it would make no sense to support `for`-loops. Therefore, the user has to "unfold" the expressions, thus generating a sequence of assignment statements and to replace the `for`-loop by the equivalent `while`-loop.

Example 4.6. *Figure 3.5 shows how a* `do-while` *loop is simulated by a conditional statement and a* `goto`*-statement. BOOP introduces a fresh label ℓ at the beginning of the body of the loop. For reasons of simplicity, the first statement in the loop body of the abstract program will always be a* `skip`*-statement. At the end of the loop, the control flow is transferred to ℓ if the weakening of the guard evaluates to* `1`*. The loop can only be left if the weakening of the negation of the guard evaluates to* `1`*.*

We noticed that in the example presented in Chapter 3 the number of iterations of the `do-while`*-loop has no influence on the reachability of the error label. Therefore, the weakening of the guard as well as the weakening of its negation is* `1`*, allowing any number of iterations.*

4.5.5 Procedures

A major advantage of the presented refinement/abstraction algorithm over other approaches that are limited to finite state machines (e.g. reachability checking with BDDs [Gor00]) is that it can handle procedure calls. This opens the possibility of model checking programs with a theoretically unrestrictedly large state space[10]. Each procedure $\mathcal{R}_{\mathbb{B}}$ in a Boolean Program has access to the global predicates E_G a finite number of local predicates $E_{\mathcal{R}}$. Thus $E_G \cup E_{\mathcal{R}}$ denotes the predicates that are in scope within the procedure $\mathcal{R}_{\mathbb{B}}$. The cardinality of the set of predicates in scope is always bounded in a Boolean Program, since neither pointers nor allocation of memory is allowed.

For a C program $\mathcal{P}$ the situation is quite different. Each procedure $\mathcal{R}$ can access the global variables $G_{\mathcal{P}}$, its local variables $L_{\mathcal{R}}$, and its formal parameters $F_{\mathcal{R}}$. It can potentially even access variables that are local to other procedures, because one of the actual parameters may be a pointer to such a location. By constructing a linked list the state space may thus become arbitrarily large, even if we do not allow allocation of additional memory via `malloc`. Memory allocation does additionally introduces the possibility of unbounded data types.

[10] The undecidability of the halting problem does of course impose some restrictions. These will be covered by Chapter 6

We provide summary of the notation that is used in this section in Figure 4.4. Note the assumption that each C procedure has a unique return variable. Though a C procedure can in general return an arbitrary expression, we assume for the sake of simplicity that the return variable is assigned this expression and then returned. This is also the way of how the GCC compiler handles return values.

Boolean Programs		
E	...	the set of predicates used for the abstraction
E_G	...	$E_G \subseteq E$, global predicates of the Boolean Program
$E_{\mathcal{R}}$	...	$E_{\mathcal{R}} \subseteq E$, $E_{\mathcal{R}} \cap E_G = \emptyset$ predicates local to procedure $\mathcal{R}_{\mathbb{B}}$.
$E_{\mathcal{R}f}$	...	$E_{\mathcal{R}f} \subseteq E_{\mathcal{R}}$, the set of formal parameter predicates of procedure $\mathcal{R}_{\mathbb{B}}$
$E_{\mathcal{R}r}$	...	$E_{\mathcal{R}r} \subseteq E_{\mathcal{R}}$ the set of return predicates of $\mathcal{R}_{\mathbb{B}}$
C Programs		
$G_{\mathcal{P}}$	...	global variables of the C program $\mathcal{P}$
$L_{\mathcal{R}}$	...	variables local to procedure $\mathcal{R}$
$F_{\mathcal{R}}$	...	formal parameters of procedure $\mathcal{R}$
r	...	return variable of procedure $\mathcal{R}$

Figure 4.4: Notation used for abstracting procedure calls

Before we explain how procedures and procedure calls can be translated to Boolean Programs, we have to define the *scope* of predicates in a Boolean Program. A predicate is in scope at a certain program point, if any of the variables that it mentions[11] are in scope at the corresponding program point in the C program. This does also include variables that can be accessed via aliasing, though this case is not covered our the current implementation.

In order to gain an abstraction of C procedures we have to calculate the new *signature* of the Boolean procedure with respect to the predicate set E. Furthermore we have to handle the *calls* that are made *to the procedure* – i.e., determine the actual parameter predicates and update the predicates in scope after the procedure has returned. Therefore, the translation of a C procedure $\mathcal{R}$ involves two steps:

1. **Calculating Signatures**. In Figure 4.4 we have already specified the elements that are needed to define a procedure in a Boolean Program. We will now give a formal definition of how they are gained [BMMR01]:

   - $E_{\mathcal{R}f}$ is the set of formal parameter predicates of $\mathcal{R}_{\mathbb{B}}$.

     $$\{e \in E_{\mathcal{R}} \mid vars(e) \cap L_{\mathcal{R}} = \emptyset\} \tag{4.11}$$

     This means that formal parameter predicates must not contain any reference to local variables. They may mention either global variables or formal parameters of the function $\mathcal{R}$.

   - $E_{\mathcal{R}r}$ is the set of return predicates of $\mathcal{R}_{\mathbb{B}}$. Note that procedures in a Boolean

[11] In C programs, local variables may have the same name as global variables, thus shadowing some of the global variables. In our internal representation of the variables, we assign a unique identifier to each variable, thus preventing problems with variables of the same name.

Program may have more than one return value (see Section 4.2).

$$E_{\mathcal{R}r} = \left\{ e \in E_{\mathcal{R}} \;\middle|\; \Big(r \in vars(e) \wedge (vars(e) \setminus \{r\} \cap L_{\mathcal{R}} = \emptyset) \Big) \vee \Big(e \in E_{\mathcal{R}f} \wedge (vars(e) \cap G_{\mathcal{P}} \neq \emptyset \vee drfs(e) \cap F_{\mathcal{R}} \neq \emptyset) \Big) \right\} \tag{4.12}$$

The first line of Equation 4.12 determines that all predicates that mention the return variable r but none of the other local variables of the procedure $\mathcal{R}$ must be returned by $\mathcal{R}_{\mathbb{B}}$. These predicates contain information about the return value of $\mathcal{R}$. Predicates that mention local variables of $L_{\mathcal{R}}$ cannot be returned, since the local variables are undefined outside $\mathcal{R}$. The second line of Equation 4.12 says that all formal parameter predicates that mention either a global variable or dereference a formal parameter of $\mathcal{R}$ must be returned. This is necessary to provide information about any global values or call-by-reference parameters that may have been changed by the procedure.

Example 4.7. *Figure 3.4(a) shows the definition of a Boolean function. There are no global predicates,* $\{inode \rightarrow i\_rdev \gg 8 = major\}$ *and* $\{testdev\_open = 0\}$ *are local to* `testdev_open`*. The signature of the concrete C function is*

```
int testdev_open (struct inode *inode, struct file *filp)
```

The variables that are local to `testdev_open` *are* `inode`*,* `filp` *and the return variable (which denote by* `testdev_open`*). We yield the set of formal predicate parameters from Equation 4.11.*

$$E_{\mathcal{R}f} = \Big\{ \{inode \rightarrow i\_rdev \gg 8 = major\} \Big\}$$

Equation 4.12 gives us the two return predicates:

$$E_{\mathcal{R}r} = \Big\{ \{inode \rightarrow i\_rdev \gg 8 = major\}, \{testdev\_open = 0\} \Big\}$$

The first predicate in the set must be returned because it dereferences a formal parameter. The second predicate mentions the return value and no local variable. Thus, we gain the signature of the abstraction of `testdev_open`*:*

```
bool<2> testdev_open ({inode→i_rdev≫ 8 =major})
```

2. **Handling Procedure Calls.**

For each call to the C procedure $\mathcal{R}(f_1, \ldots, f_n)$ (where $\{f_1, \ldots, f_n\} = F_{\mathcal{R}}$) with the actual parameters $a_1, \ldots, a_n$ we must determine the actual parameter predicates for the corresponding call to the Boolean procedure $\mathcal{R}_{\mathbb{B}}(e_1, \ldots, e_m)$ (where $\{e_1, \ldots, e_m\} = E_{\mathcal{R}f}$). For this purpose, we have to substitute the actual parameters $a_1, \ldots, a_n$ for the actual parameters $f_1, \ldots, f_n$ in the formal parameter predicates:

$$e_i' = e_i[a_1/f_1, \ldots, a_n/f_n] \tag{4.13}$$

This substitution can be understood as translating the formal parameter predicates of $\mathcal{R}_{\mathbb{B}}$ to the *calling context*. This is necessary, since the formal parameters have no

interpretation outside $\mathcal{R}_{\mathbb{B}}$. The substitution yields a set of actual parameter predicates that is in general not a subset of the predicate set E. Again, we have to approximate the set of states described by these formal parameter predicates in terms of the predicates in $E_G \cup E_{\mathcal{S}}$. (Here, $\mathcal{S}$ is the *calling* procedure, i.e. the procedure that contains the label ℓ where $\mathcal{R}$ is called. $E_G \cup E_{\mathcal{S}}$ are therefore the predicates in scope at the calling point.) We must not overestimate the set of states that the predicates $e'_1, \ldots, e'_n$ represent. Therefore we *strengthen* the predicates to expressions over the predicates in $E_G \cup E_{\mathcal{S}}$. The actual parameter for the formal e_i is then

$$\texttt{choose}(\mathcal{F}(e'_i, E_{\mathcal{G}} \cup E_{\mathcal{S}}), \mathcal{F}(\neg e'_i, E_{\mathcal{G}} \cup E_{\mathcal{S}})) \tag{4.14}$$

and the procedure call in the Boolean Program will be

$$\begin{aligned} t_1, \ldots, t_k \;:=\; \mathcal{R}_{\mathbb{B}}(&\texttt{choose}(\mathcal{F}(e'_1, E_G \cup E_{\mathcal{S}}), \mathcal{F}(\neg e'_1, E_G \cup E_{\mathcal{S}})),\\ &\ldots,\\ &\texttt{choose}(\mathcal{F}(e'_m, E_G \cup E_{\mathcal{S}}), \mathcal{F}(\neg e'_m, E_G \cup E_{\mathcal{S}}))); \end{aligned} \tag{4.15}$$

where $t_1, \ldots, t_k$ denote fresh variables with k being the cardinality of $E_{\mathcal{R}r}$ and $e'_1, \ldots, e'_m$ are the formal parameter predicates translated to the calling context (as defined above).

C programs allow only one return value. Assume that this return value is assigned to v, i.e. the call to the C procedure is $v := \mathcal{R}(a_1, \ldots, a_n);$. Such a call potentially changes the value of all (global and local) predicates that mention v (either directly or indirectly). Additionally, any predicate in $E_{\mathcal{S}}$ that mentions a global variable, a dereference of an actual parameter, or indirectly (via pointers) mentions either of these kinds of locations may have changed as result of the procedure call and must be updated[12].

Let E_u denote the set of predicates that must be updated after a call to $\mathcal{R}_{\mathbb{B}}$. The remaining predicates ($E_{\mathcal{S}} \cup E_G$) along with the return value predicates $E_{\mathcal{R}r}$ are used to update the predicates in E_u. Any occurrences of predicates out of $E_{\mathcal{R}r}$ in the strengthened expression must be replaced by the corresponding element out of $\{t_1, \ldots, t_k\}$, since the predicates in $E_{\mathcal{R}r}$ are undefined outside $\mathcal{R}_{\mathbb{B}}$.

Since we want to update predicates outside of $\mathcal{R}_{\mathbb{B}}$ we have to translate the predicates in $E_{\mathcal{R}r}$ to the *calling context*:

$$\text{For each } e_i \in E_{\mathcal{R}r},\ e'_i = e_i[v/r, a_1/f_1, \ldots, a_n/f_n] \text{ and } E'_{\mathcal{R}r} = \{e'_1, \ldots, e'_k\} \tag{4.16}$$

Each e'_i is associated with the corresponding t_i. Now, each $e \in E_u$ is assigned the value

$$\begin{aligned} \texttt{choose}(&\mathcal{F}(e, (E_G \cup E_{\mathcal{S}} \cup E'_{\mathcal{R}r}) \setminus E_u),\\ &\mathcal{F}(\neg e, (E_G \cup E_{\mathcal{S}} \cup E'_{\mathcal{R}r}) \setminus E_u)) \end{aligned} \tag{4.17}$$

with the t_i substituted for the corresponding e'_i.

[12] Our implementation does not cover the case that either global variables or (possibly transitive) dereferences of actual parameters are mentioned via pointers, since it does not yet contain a pointer analysis algorithm. In the SLAM toolkit, a pointer analysis is used to determine a conservative over-approximation to the set of predicates to update [BMMR01]

Example 4.8. *Figure 3.3 and Figure 3.5 show a call to* `testdev_open` *at label* $\mathcal{B}$*. We calculated the boolean signature of this function in Example 4.7. We will now show how a call to this function is handled.*

The only formal parameter of the function is $e = \{inode \rightarrow i\_rdev \gg 8 = major\}$*. A translation of this predicate to the calling context yields* $e' = \{(*(\&inode)).i\_rdev \gg 8 = major\}$ *(where the variable* `inode` *in* e' *denotes a different location than in* e*, since the variable* `inode` *that we substituted for the formal parameter is not in the scope of* `testdev_open`*). There exist no predicates outside* `testdev_open` *that talk about the value of* `inode`*, therefore, the strengthening of the predicate in the calling context can only yield* `0`*. The value of the formal parameter predicate is therefore set non-deterministically.*

Translating the return predicates of the function to the calling context yields the predicates $\{(*(\&inode)).i\_rdev \gg 8 = major\}$ *and* $\{rval = 0\}$*. The former predicate has no influence on the predicates that are in the scope of the calling function. The latter predicate, however, has an exact equivalent. Therefore, the predicate* $\{rval = 0\}$ *must be updated. It is assigned the return value:*

$\{rval = 0\}$ `:= choose (`ret_2`,!`ret_2`);`

The abstract code of `testdev_open` *in Figure 3.4(b) modifies the global predicate* $\{locked \neq 0\}$*. However, since no global predicate mentions the variable* `rval`*, there is no need to update any global predicates.*

4.5.6 Concerning Labels

As indicated in Section 4.5.3, labels are taken over unmodified. Additionally, we introduce a fresh label for each line in the Boolean Program. This is necessary to map paths that the model checker provides as counterexample back to the original C program. Such a label usually consists of a prefix, the file name[13], the line number, and a unique identifier. (The latter is necessary because a line in a C program may contain several statements).

[13] At the moment, the directory names are truncated. This may lead to problems in big projects where duplicate filenames occur in the different directories.

Chapter 5

Formalization of C expressions in HOL98

5.1 Introduction

In the previous chapter we have assumed that there is a decision procedure that is able to decide if a C expression is either true or false. Such a procedure is necessary for weakening and strengthening of preconditions (as explained in Section 4.4) and for determining the feasibility of a path (covered by Chapter 7). However, most theorem provers (including HOL98) are based on first order logic or higher order logic and do not accept C expressions as input.

In order to be able to attempt formal proofs on C expressions we have to make rigorous assumptions about the semantics of the C programming language. The C expressions must be transformed to the logic of the theorem prover.

Due to our decision for the Theorem Prover HOL98 we will give a definition of the semantics of C expressions in terms of the logic underlying the HOL system (as specified in [Uni01a]). In this chapter, we assume that the reader is familiar with the HOL98 Theorem Prover. A recommendable introduction can be found in [Uni01b].

Following the notation used in [Sch86] we define a translation function $[\![\text{-}]\!]$ for expressions and $\langle\!|\text{-}|\!\rangle$ for types. The former function maps C expressions to HOL expressions, while the latter function maps C types to HOL types. This Chapter covers the complete subset of expressions that can be handled by the Theorem Prover module at the current state of the implementation. The covered parts of the type system of C are formalized in Section 5.3. Section 5.6 covers all kinds of operations, including pointer arithmetic (Section 5.6.3).

Furthermore, several constructs that may occur in C expressions have no equivalent in higher order logic. This chapter covers in detail how these constructs are translated.

Some of our assumptions about the semantics of C expressions may lead to a model that is inappropriate for certain platforms or compilers. We cannot overcome this problem unless we restrict our considerations to a certain compiler on a specific platform. Even then it might be hard to model every detail of the concrete semantics in terms of a formal system. We will therefore content ourselves with an approximate (but still rigorous) definition of the meaning of C expressions. Admittedly, this introduces the possibility of either spurious or missing counterexamples. However, this will only be the case if the programmer makes use of platform- or compiler-dependent properties of the C programming language (which should be discouraged in favor of portability). The restrictions we made are presented in Section 5.2.

At the end of this chapter, we provide a brief overview of our decision procedure, and we explain, why there are C expressions that it cannot decide (as mentioned at the end of Section 4.4.4).

5.2 Restrictions

We had to make several restrictions to the set of C expressions that can be handled by the Theorem Prover. Some of them are quite natural and can be justified by the different interpretations of the notion of an *expression* inherent to the C programming language and the HOL system. Others are due to the limited (time) resources that were available for this master's thesis. For restrictions of the latter nature we will give some suggestions of how our program can be extended to cover those cases.

5.2.1 Expressions with Side Effects

Expressions with side effects (like `(a == ++b)`) are an element that is heavily used by many programmers (even though such constructs often impair readability). In higher order logic there is no equivalent to expressions with side effects[1]. Instead of attempting to model this property of C programs, we restrict the set of expressions we can handle to *side effect free* expressions. Note that an expression with side effects can always be split up into a sequence of statements that is executed before the evaluation of the expression, a side effect free expression, and a sequence of statements that follow the evaluation. Though such a split is in general not hard to accomplish, it is not yet part of our implementation. At the moment, it is task of the programmer to keep the expressions in his program free of side effects.

5.2.2 Pointer arithmetic

Though our implementation of the predicate transformer $\mathcal{WP}$ does not handle pointers at the moment (see Section 4.3.1) the Theorem Prover can handle at least a subset of the expressions that contain pointer arithmetic. However, if a memory location is accessed by dereferencing a constant value or by dereferencing an expression like `(&a+10)` we cannot make any assumptions on the result of such an operation (unless a value was assigned to exactly this location, as in $(*(0\text{xA50}) = 10 \wedge b = 0\text{xA50}) \implies *b < 20)$). Since we have no knowledge of where the objects exactly reside in memory (this may be different for each execution of the program) we must assume that such an expression can reference an arbitrary object. Therefore, such constructs must be avoided.

5.2.3 Bit manipulation operators

At the current state of our implementation the only supported bit manipulation operators are *shift right* ($>>$) and *shift left* ($<<$). Integers are represented by the standard interpretation of $\mathbb{N}$ as defined in HOL98. The `integer` Theory of HOL98 does not provide bit manipulation operators. We have added a new (but yet incomplete) theory to HOL98 that defines the above mentioned operators in terms of standard operations for $\mathbb{N}$. This theory will be described in detail in Section 5.6.1.

[1] Furthermore, the predicate transformer $\mathcal{WP}(S, \varphi)$ is not defined for expressions φ with side effects.

5.2.4 n-bit integers

The range of values that integer variables in C programs may take is bounded by the bit size of the variable. Integer variables in HOL do not have this property. Still, the usage of $\mathbb{N}$-typed variables for representing n-bit integers does not introduce an inaccuracy; It is the semantics of the operations defined in the `integer` theory of HOL98 that does not adhere to the same rules as the integer operations in C programs. This problem could be overcome by using a modulo-arithmetic. The `word` theory of HOL98 would fit such purposes. Alternatively, we could check for integer overflows explicitely. However, we found it more natural to make use of the `integer` theory, because this allows us to use the powerful decision procedures of this theory. A similar approach has been chosen in the SLAM project.

5.2.5 Floating Point Arithmetic

Floating point types in C programs have restricted accuracy. Elements of type $\mathbb{R}$ in the `real` Library of HOL98 do not adhere to this restriction. While the set of float values in C programs is recursively enumerable, the set $\mathbb{R}$ is non-enumerable. This fact is neglected in our implementation, since we do not expect the approach to yield satisfying results for C programs that make heavy use of floating point arithmetic. An exact formalization of C floating point types would lead to complicated HOL terms that cannot be decided by our decision procedure.

5.3 Types

As mentioned in Chapter 2, higher order logic is a strongly typed formalism. Therefore, we need to translate C types to HOL types. This requires a consistent type system for our formalization of C expressions. At the current state of our implementation, we support following C types:

1. **Booleans**. ANSI C does not support Boolean types. However, we still provide the possibility to handle expressions that contain Boolean types, since they may be introduced by our wrapper routines that transform the GCC abstract syntax tree to our internal representation of C programs, e.g. when binary expressions involving conjunction or disjunction are translated. In HOL we are using the type $\mathbb{B}$ of the `bool` theory.

2. **Integer types**. The ANSI C standard defines the integer types `char` and `int`. Both may either be signed or unsigned, and their bit length may be varied with help of the modifiers `long` and `short`. The only restrictions that the ANSI C standard makes on these types are that `short` and `int` variables have at least 16 bits, `long` at least 32 bits, and that the range of `short` values is not larger than the range of `int` values, and that the range of `int` values must not be larger than the range of `long` values [KR88]. Since the size of these integral types varies from platform to platform (and for other reasons, as mentioned in Section 5.2.4) we chose to translate all integer types to the $\mathbb{Z}$ type of the `integer` theory of HOL.

$$\begin{aligned} \langle\!| \text{[signed|unsigned] [short|long] char} |\!\rangle &= \mathbb{Z} \\ \langle\!| \text{[signed|unsigned] [short|long] int} |\!\rangle &= \mathbb{Z} \end{aligned} \tag{5.1}$$

3. **Real types**. As discussed in Section 5.2.5 representing C floating point types (`real` and `float`) as $\mathbb{R}$ is not completely accurate. Rounding errors are not reflected if we translate floating point values to elements of $\mathbb{R}$. A completely accurate formalization of IEEE floating point arithmetic would have been beyond the scope this thesis. Furthermore, we would not have been able to use the decision procedures that HOL98 provides for real arithmetic.

$$\begin{aligned} \langle\!|\text{float}|\!\rangle &= \mathbb{R} \\ \langle\!|[\text{long}]\ \text{double}|\!\rangle &= \mathbb{R} \end{aligned} \tag{5.2}$$

4. **Enumeration constants**. We handle enumeration constants similar to integer constants.

$$\langle\!|\text{enum}\ \{\ldots\}|\!\rangle = \mathbb{Z} \tag{5.3}$$

   We use $\mathbb{Z}$ to represent enumeration constants, though $\mathbb{N}$ would be more intuitive. However, since most operators are only defined for $\mathbb{Z}$ and $\mathbb{R}$, this decision prevents the introduction of type casts from $\mathbb{N}$ to $\mathbb{Z}$, which would be introduced whenever enumeration constants occur in expressions.

5. **Pointer types**. Pointer arithmetic is a property of C programs that is not reflected in any standard logic as higher order logic. Though the values of pointers can be represented as elements of $\mathbb{N}$, the problem remains that the value that is referenced by the pointer may be of arbitrary type. This yields the necessity of *wrapping* and *unwrapping* the elements that are referenced by a pointer. A detailed description of this approach will be given in Section 5.6.3. For the time being, we content ourselves with the fact that natural numbers are a suitable way of representing pointers.

$$\langle\!|\sigma*|\!\rangle = \mathbb{N} \tag{5.4}$$

   Here, $\sigma*$ denotes a pointer to an arbitrary type σ.

6. **Compound types**. Compound types denote either `structs` or `unions`. We handle both types the same way, though this may lead to inaccuracies if the programmer tries to exploit the mechanism of how variables of type `union` are stored in memory. This, however, is a platform dependent property. Schmaranz recommends to avoid the `union` type unless there is a compelling reason to use it [Sch02]. Though HOL supports compound types (resp. records) and this would be the obvious way to represent C structures in HOL, we have decided to use a different encoding for reasons of simplicity[2]. We represent any structure object as natural number and access its fields with help of *projection functions*. There are several ways of unambiguously encoding an n-tuple of values in one element of $\mathbb{N}$ (one would be to use products of prime numbers, e.g. $(n, m, r, \ldots) \equiv 3^n \cdot 5^m \cdot 7^r \cdot \ldots$ [BBJ02]). By representing the above mentioned projection functions as function variables, we leave the exact field access mechanism open and just specify enough of their properties to restrict the number of possible models to those that fit our purpose (see Section 5.6.4).

$$\begin{aligned} \langle\!|\text{struct}\ \{\ldots\}|\!\rangle &= \mathbb{N} \\ \langle\!|\text{union}\ \{\ldots\}|\!\rangle &= \mathbb{N} \end{aligned} \tag{5.5}$$

[2]HOL records need to be defined explicitely before they are used. This would require us to define HOL records for all compound types that occur in the C expressions, resulting in additional calls to the theorem prover. The implementation would have been more complicated than the implementation of the approach that we use at the moment

7. **Field types**. The type of a field in a structure is simply the type of the specific element as given in the definition of the `struct` type. (We find this fact worthy of being noted, since the GCC uses a separate type for fields of compound types).

5.4 Constants

Constants are an essential element of C expressions. We provide a description of all constants that we support at the moment.

1. **Boolean constants**. Though Boolean constants are not directly supported by ANSI C compilers, we provide a representation for them, since Boolean constants may be introduced by the functions that are responsible for the calls to the theorem prover (e.g., cubes that contain no predicates are represented by the boolean constant "true". The `bool` theory of HOL provides the two constants **F** (for false) and **T** (for true), of which we make use.

2. **Integer constants**. Though the integer types of C and HOL differ in the cardinality of their ranges, we use the integer representation of HOL for the representation of C integer constants. The `integer` theory of HOL does not provide constants for each integer value, since this would result in an infinite signature of the theory. Instead, integer constants are decomposed into bits and expressed with help of constructor functions [Uni01a]. This mechanism is hidden by the parser of the HOL theorem prover.

3. **Real constants**. HOL does not support real constants directly. They must be constructed using quotients of integers instead [Uni01a] (e.g., the floating point constant 123.4 must be represented as 1234/10). Our current implementation does not support real constants.

4. **Compound constants and other constants**. At the current state of our implementation, Boolean and integer constants are the only supported constant values. Note that enumeration constants are handled similar to integer constants.

5.5 Variables

Any occurrence of a C variable `a` is simply translated to a HOL variable of the corresponding type (see Section 5.3). In our implementation, an unique identifier is appended to every variable name. This is necessary to prevent the confusion of variables that are defined in different scopes.

5.6 Operations

Most of the C operations that are supported by our current implementation are mapped to their equivalent operations in HOL in a natural way. Some operations (like bit manipulation operations) do not have equivalent operations in the HOL98 theories that we use. In this case it was necessary to add a theory that defines the missing operations in terms of existing ones. This theory will be described in Section 5.6.1 together with the bit manipulation operations.

In ANSI C any operator can be used on virtually any type. Whenever the type of an operand does not fit for the operator that is used, the compiler implicitly introduces a *type cast* (this operation will be described in the section that covers the unary operations). However, the usage of logical operators like `&&` and `||` must be handled separately, since the notion of a Boolean type does not exists in ANSI C - a C expression is assumed to be true if its evaluation yields a value different from zero. This means, that e.g. `5 && (a == b)` is a perfectly valid C expression. HOL, with its rigorous type system, does not allow such constructs. A negated comparison of such "integral" Boolean values to 0 yields a Boolean expression that can then be translated to the HOL logic (e.g., $[\![$`5 && (a == b)`$]\!] = \neg(5 = 0) \wedge (a = b)$). From here on, we do not mention this conversion anymore, and we assume that it is applied whenever necessary.

5.6.1 Binary Operations

The types of the binary operations that define here have the structure $\sigma_1 \times \sigma_1 \rightarrow \sigma_2$ (the shift operations are an exception). This means that expressions like $(5.1 < 3)$ are not supported at the moment, since type casts are not completely implemented yet (type casts are described together with the unary operations).

The HOL system supports overloading of operations, what means that two different operations constants may be denoted by the same symbol [Uni01a]. This mechanism does not introduce ambiguity, since the operations can still be kept apart by their signature[3]. Our implementation uses the non-overloaded names of the operations. In this section we favor the usage of the overloaded (and pretty-printed) names for reasons of simplicity. The translation function inserts the proper operation depending on the types of the type of the left and the right operator.

Following binary expressions are supported by our implementation:

1. **Logical Disjunction and Conjunction**. Conjunction and Disjunction is translated to the corresponding operations of the HOL `bool` theory.

$$\begin{aligned} [\![e_1 \;\&\&\; e_2]\!] &= [\![e_1]\!] \wedge [\![e_1]\!] \\ [\![e_1 \;||\; e_2]\!] &= [\![e_1]\!] \vee [\![e_1]\!] \end{aligned} \tag{5.6}$$

2. **Comparisons**. The expressions $[\![e_1]\!]$ and $[\![e_2]\!]$ can be either be of type $\mathbb{R}$ or of type $\mathbb{N}$. In the following definitions, we use the overloaded names for the HOL operations:

$$\begin{aligned} [\![e_1 == e_2]\!] &= [\![e_1]\!] = [\![e_1]\!] \\ [\![e_1 \;!= e_2]\!] &= \neg([\![e_1]\!] = [\![e_1]\!]) \\ [\![e_1 > e_2]\!] &= [\![e_1]\!] > [\![e_1]\!] \\ [\![e_1 < e_2]\!] &= [\![e_1]\!] < [\![e_1]\!] \\ [\![e_1 <= e_2]\!] &= [\![e_1]\!] \leq [\![e_1]\!] \\ [\![e_1 >= e_2]\!] &= [\![e_1]\!] \geq [\![e_1]\!] \end{aligned} \tag{5.7}$$

[3] A *signature* consists of the operator name and the operator type, e.g. $\leq$ can either be of type $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$ or of type $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{B}$, thus representing the two different operations `int_le` and `real_le`.

3. **Arithmetic Operations**. The remarks regarding overloaded operations that were made with respect to comparisons are valid for arithmetic operations, too. However, note that there is no modulo operation for $\mathbb{R}$.

$$\begin{aligned} [\![e_1 + e_2]\!] &= [\![e_1]\!] + [\![e_1]\!] \\ [\![e_1 - e_2]\!] &= [\![e_1]\!] - [\![e_1]\!] \\ [\![e_1 * e_2]\!] &= [\![e_1]\!] \cdot [\![e_1]\!] \\ [\![e_1 / e_2]\!] &= [\![e_1]\!] / [\![e_1]\!] \\ [\![e_1 \% e_2]\!] &= [\![e_1]\!] \bmod [\![e_1]\!] \end{aligned} \tag{5.8}$$

4. **Bit Manipulation Operations**. At the moment, the only supported operations are *shift left* and *shift right*. The effect of applying the *shift right operation* to signed values may be a shift with insertion of either the sign bit or the null bit on the left side. The concrete semantics is implementation dependent and may vary from compiler to compiler. Even the same compiler might yield different results on different platforms. If we consider the *shift right operation* and apply it to a signed char value as follows

$$((\text{signed char}) - 1) >> 8$$

we get an expression whose truth value depends on the compiler and the platform that is used[4].

Since none of the HOL Theories we use contains operations that can be used for manipulating single bits we had to add a theory that defines such operations in terms of existing operations. The Theorem `SHIFTLEFT_THM` provides a recursive definition of the *shift left* operation:

$$\begin{aligned} &\text{SHIFTLEFT } (v : \mathbb{Z})\ (0 : \mathbb{N}) = v \\ &\text{SHIFTLEFT } (v : \mathbb{Z})\ (\text{SUC } n : \mathbb{N}) = \text{SHIFTLEFT } (v + v)\ n \end{aligned} \tag{5.9}$$

Here, `SUC` denotes the successor function for natural numbers. By taking n as measure we can show that the recursive definition yields a terminating rewriting rule (from left to right) [BN98, Uni01a]. Thus, the theorem prover is able to replace all occurrences of `SHIFTLEFT` by a series of additions. Obviously we have chosen to maintain the sign of the left operand. This is not compliant with the ANSI C standard, but yields a very simple definition of the shift left operation. A more complex definition would result in expressions that are harder to decide for the theorem prover. Note, that this inaccuracy may lead to incorrect results.

The `SHIFTRIGHT` function is defined similar by using division by 2 instead of addition. Now the *shift* operations can be translated as follows:

$$\begin{aligned} [\![e_1 << e_2]\!] &= \text{SHIFTLEFT } [\![e_1]\!]\ \text{Num}([\![e_2]\!]) \\ [\![e_1 >> e_2]\!] &= \text{SHIFTRIGHT } [\![e_1]\!]\ \text{Num}([\![e_2]\!]) \end{aligned} \tag{5.10}$$

where `Num`: $\mathbb{Z} \to \mathbb{N}$ denotes the (partial) function that maps integer values to natural numbers.

[4] This is partly because of the loose specification of the shift right operation, but also because the bit size of the signed char type may vary from platform to platform – though this is usually not the case.

Example 5.1. *"$i << 3$" will be translated to "SHIFTLEFT i Num(3)". HOL98 rewrites this expression to "$i + i + i + i + i + i + i + i$" (what is equivalent to $8 \cdot i$).*

Furthermore, we added some supportive theorems for the rewriting tactics of the theorem prover:

$$\begin{aligned} \forall a\ n.\ \text{SHIFTRIGHT (SHIFTLEFT } a\ n)\ n = a \\ \forall a\ n.\ (\text{SHIFTLEFT } (2 \cdot a)\ n)/2 = \text{SHIFTLEFT } a\ n \end{aligned} \tag{5.11}$$

The HOL system ensures that only well-formed theories can be constructed by allowing theorems to be created only by formal proof [Uni01a]. The theorems defined above were proved using induction on n.

5.6.2 Unary Operations

This section covers all unary operations that we support at the moment. Unary operations include type casts and address and pointer operations, too.

1. **Negations**. Logical and Arithmetic Negations are translated the natural way. Note that the minus prefix is polymorphic and denotes different operations for $\mathbb{Z}$ and $\mathbb{R}$.

$$\begin{aligned} [\![!e_1]\!] &= \neg[\![e_1]\!] \\ [\![-e_1]\!] &= -[\![e_1]\!] \end{aligned} \tag{5.12}$$

2. **Type Casts**. As mentioned above, C compilers implicitly introduce type casts whenever necessary. At the moment we have implemented only rudimentary support for type casts. Since we represent integer types of all sizes as elements of $\mathbb{Z}$ (no matter if they are signed or unsigned) and all floating point values as elements of $\mathbb{R}$, all conversions that would not change the type of the expression in the HOL expression that results of [![-]!] are ignored. In the remaining cases, which are not yet supported, the operators `int_of_num` and `real_of_num` of the `integer` resp. the `real` theory of HOL98 might be useful for representing type casts.

$$\begin{aligned} [\![(\text{[signed|unsigned] [short|long] char|int})ival]\!] = ival,\ \text{if } [\![ival]\!] \text{ is of type } \mathbb{Z} \\ [\![(\text{[long] double|float})rval]\!] = rval,\ \text{if } [\![rval]\!] \text{ is of type } \mathbb{R} \end{aligned} \tag{5.13}$$

3. **Address and Pointer Operations**. The translation of the unary operations `&` and `*` will be explained in Section 5.6.3.

5.6.3 Pointer arithmetic

Despite the restrictions that were mentioned in Section 5.2.2 our current implementation of the translation function [![-]!] can handle most expressions that contain pointers (resp. the unary operations `&` and `*`). There is no "natural" way of translating C pointers to Higher Order Logic, since the notion of a *location* does not exists in HOL. Furthermore, the usage of pointers indirectly introduces a type polymorphism by allowing `void*` pointers. When referencing and dereferencing variables with help of the operations `&` and `*` the initial type

must not be changed (i.e., `*(&a)` must have the same type as `a`). Therefore, we introduced a new compound datatype for the representation of pointer values. As explained in Section 5.3, C types are translated to either $\mathbb{Z}$, $\mathbb{R}$ or $\mathbb{N}$. The datatype `loc` wraps the original type of the referenced variable.

$$\begin{aligned} \text{Hol\_datatype loc} \;=\; & \mathbb{Z}\text{-loc of } \mathbb{Z} \\ \mid\; & \mathbb{R}\text{-loc of } \mathbb{R} \\ \mid\; & \mathbb{N}\text{-loc of } \mathbb{N} \end{aligned} \tag{5.14}$$

Additionally, we have defined one unwrapping function for each type that can be wrapped up in a `loc`-type.

$$\begin{aligned} \text{unrep-}\mathbb{Z}\ (\mathbb{Z}\text{-loc } i) &= i \\ \text{unrep-}\mathbb{R}\ (\mathbb{R}\text{-loc } r) &= r \\ \text{unrep-}\mathbb{R}\ (\mathbb{R}\text{-loc } n) &= n \end{aligned} \tag{5.15}$$

To represent the "address of" and "points to" operations, we introduce a new function variable ptr: $\mathbb{N} \rightarrow$`loc` that will be used to model the dereferencing operation `*`. We just specify enough of its properties to restrict the number of possible models to those that fit our purpose.

1. **Referencing Variables**. If referencing a variable `a` with help of the operation `&` yields an address ℓ_a we have to guarantee that $ptr(\ell_a) = a$. We add this information by supplying a *premise* that states that the pointer function maps the address we gained as result of the translation function to a.

$$[\![\&\mathrm{a}]\!] = \begin{cases} \text{unrep-}\mathbb{Z}\ (ptr(addr\_a)) = [\![a]\!] \implies \ldots addr\_a \ldots & \text{if } [\![\mathrm{a}]\!] \text{ is of type } \mathbb{Z} \\ \text{unrep-}\mathbb{R}\ (ptr(addr\_a)) = [\![a]\!] \implies \ldots addr\_a \ldots & \text{if } [\![\mathrm{a}]\!] \text{ is of type } \mathbb{R} \\ \text{unrep-}\mathbb{N}\ (ptr(addr\_a)) = [\![a]\!] \implies \ldots addr\_a \ldots & \text{if } [\![\mathrm{a}]\!] \text{ is of type } \mathbb{N} \end{cases} \tag{5.16}$$

   Here, $addr\_a$ is a variable name that uniquely represents the address of the variable a. This variable name is constructed by the translation function and is the resulting value that is associated with `&a`. The translation works similarly for fields of structures, but not for arbitrary expressions, since a C expression does in general not have an address.

2. **Dereferencing Pointers**. Since the properties of the function ptr are specified in the premise of the HOL expression, we can make use of it in the conclusion.

$$[\![{*}e_1]\!] = \begin{cases} \text{unrep-}\mathbb{Z}\ (ptr([\![e_1]\!])) & \text{if } [\![e_1]\!] \text{ is of type } \mathbb{Z}\text{-loc} \\ \text{unrep-}\mathbb{R}\ (ptr([\![e_1]\!])) & \text{if } [\![e_1]\!] \text{ is of type } \mathbb{R}\text{-loc} \\ \text{unrep-}\mathbb{N}\ (ptr([\![e_1]\!])) & \text{if } [\![e_1]\!] \text{ is of type } \mathbb{N}\text{-loc} \end{cases} \tag{5.17}$$

Example 5.2. *If* `a` *is of type integer, then the translation of* `p == &a && *p + 1 == 5` *yields*

$$\texttt{unrep-}\mathbb{Z}\ (ptr(addr\_a)) = a \implies (p = addr\_a) \wedge (\texttt{unrep-}\mathbb{Z}\ (ptr(p)) + 1 = 5) \tag{5.18}$$

The formalization we chose forced us to split the HOL expression into a *premise* and a *conclusion*. This fact is not reflected elsewhere in this chapter, since the only affected operations are `&` and `*`. The translation of all other constructs will only contribute to the conclusion, not to the premise.

5.6.4 Field access

In order to access fields of compound types (`structs` or `unions`) we introduce a new function variable that represents a projection function for each field of a compound type. These functions can be understood as partial functions that map the corresponding fields of the instances of the compound type to their values. We do not specify *how* this mapping is accomplished (since different compilers might use different strategies for storing the fields in memory, and even the same compiler might support several approaches for optimization purposes). This does of course mean, that accessing fields via pointer arithmetic is bound to fail. However, this is no serious disadvantage compared to the SLAM approach, since the pointer analysis [Das00] algorithm that was used in the SLAM toolkit [BR01] assumes that `*(p+i)` and `*p` point to the same object, no matter which value `i` takes.

The projection function variable must have a name that uniquely identifies the field of the structure that is accessed. This is accomplished by constructing the function name of the name of the compound type[5], the field name and a prefix.

$$[\![e_1.\text{fieldName}]\!] = proj\_typeName\_fieldName([\![e_1]\!]) \tag{5.19}$$

Example 5.3. *Given a compound value* `inode` *defined as*

```
struct inode {
   kdev_t i_rdev;
   ...
} inode;
```

the translation of the C expression `inode.i_rdev == 5` *yields*

$$proj\_inode\_i\_rdev(inode) = 5 \tag{5.20}$$

5.7 Decision Procedure

Instead of providing our own decision algorithms we make use of a composition of the powerful *Tactics* [Uni01a] that come with the HOL98 theorem prover. First we try to simplify the expression using the `FULL_SIMP_TAC` of `simpLib`. This tactic applies a set of rewriting rules to the expression until no more simplifications can be made. The set of rewriting rules we are using is `hol_ss`, which includes theorems appropriate for lists, pairs, and arithmetic, combined with a number of integer reduction rules and the theorems we added to handle the shift operations and pointers. Whenever this decision procedure failed in our tests, we tried to extend the set of rewriting rules by a number of appropriate rules from the `integer` theory. If this simplification approach does not yield either true or false, the `COOPER_TAC` tactic is applied to the simplified expression. This tactic is particularly good at deciding expressions of *Presburger Arithmetic* [End72]. If the composition of `FULL_SIMP_TAC` and `COOPER_TAC` fails to decide if an expression is valid or not, it is very likely that the expression contains floating point arithmetic. In this case, we alternatively apply `REAL_ARITH_TAC` from the `real` Library of HOL98.

[5] Anonymous types are yet not supported.

The decision procedure we described above is far from being perfect. A desirable extension would be a "lightweight" Nelson-Oppen approach[6], i.e. a congruence closure algorithm [Hur01] that reduces the number of (or ideally eliminates all) occurrences of the function variables that we introduced to represent pointers and field projections. The function applications could be replaced by fresh scalar variables, e.g. $proj\_inode\_i\_rdev(inode_1) = proj\_inode\_i\_rdev\_inode_1$ and $proj\_inode\_i\_rdev(inode_2) = proj\_inode\_i\_rdev\_inode_2$. Then $inode_1 = inode_2$ would imply that $proj\_inode\_i\_rdev\_inode_1 = proj\_inode\_i\_rdev\_inode_2$. This can be done repeatedly until no more function variables occur in the expression. Of course, the expression would be exponentially bloated. An expression simplified to such an extent could be easily decided by the `COOPER_TAC` tactic, as long as the only operations used are those that are part of Presburger Arithmetic.

Structure	**Theory**
$(\mathbb{N})$	Decidable. Not finitely axiomatizable.
$(\mathbb{N}, 0, S, <)$	Decidable. Finitely axiomatizable.
$(\mathbb{N}, 0, S, <, +)$	Decidable (Presburger)
$(\mathbb{N}, 0, S, <, +, \cdot)$	Undecidable.

Figure 5.1: Decidability of certain structures [End72]

We can of course not expect to find a *complete* decision procedure that can decide all expressions that occur in our abstraction/refinement approach. This is because of the undecidability of number theory of natural numbers with multiplication[7] (see [End72, BBJ02, Göd31] and [Smu92]). Figure 5.1 gives an overview over the properties of some structures in number theory (where S denotes the successor function).

The incompleteness of the decision procedure effects our refinement/abstraction approach in so far that we cannot expect to yield useful results for programs that make heavy use of multiplication (or other operations that can only be defined in terms of multiplication). Furthermore, we are note able to apply highly complex decision procedures, since the time consumed by calls to the theorem prover has crucial influence to the over-all runtime of our tool.

5.8 Conclusions

Our current formalization of C expressions and the implementation of the decision procedure leave a lot of room for optimizations, since our principal aim was to provide a simple interface to the theorem prover without spending too much time on the implementation. For example, the use of HOL records for formalizing C structures would be much more intuitive than the approach we are using at the moment. Such a modification would very likely have positive impacts on the efficiency of the decision procedure. Another approach that might be worthwhile trying would be the use of the `word` theory of HOL98, which provides a formalization of bounded integer values.

[6] This term was coined by Konrad Slind, who was a great help whenever we had questions regarding the HOL98 theorem prover

[7] In fact, if number theory was finitely axiomatizable then the halting problem would be decidable

A translation of C expressions to propositional logic would make our expressions decidable with help of Binary Decision Diagrams or SAT solvers. In practice, it is questionable if the approach is feasible: Though Propositional Logic is decidable in theory, the decision procedures are not efficient enough to handle large formulæ. A translation of a bounded subset of the specification language VDM-SL to BDDs that we implemented as part of the PROSPER project turned out to be infeasible for larger specifications [Wei01].

Furthermore, caching the results of the theorem prover would prevent that work is repeated. Finally, it would be desirable to evaluate the suitability of other theorem provers for our approach. The theorem provers Simplify, Vampyre, and PVS were used in the SLAM project [BR01].

Chapter 6

The model checker MOPED

This chapter provides a brief description of the basic concepts of the model checker MOPED. We explain how Boolean Programs can be represented as *pushdown systems* and give a rough idea why reachability in such systems is decidable.

6.1 Pushdown Systems

Boolean Programs are equivalent to pushdown systems. A pushdown system is essentially a finite automation with a stack. The stack is used to store symbols, and only the symbols at the top of the stack are accessible. We give a formal definition of pushdown systems:

Definition 6.1. *[EHRS00] A pushdown system $\mathcal{P}$ is a system (P, Γ, Δ) where*

1. *P is a finite set of* control locations*,*
2. *Γ is a finite* stack alphabet*, and*
3. *$\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of* transition rules*.*

If $((q, \gamma), (q', \omega)) \in \Delta$ then we write $\langle q, \gamma \rangle \hookrightarrow \langle q', \omega \rangle$.

Notice that pushdown systems have no input alphabet. We are only interested if a certain *configuration* of a pushdown system representing a *state* of a Boolean Program is reachable. By a configuration of a pushdown system we denote a control location and the symbol at the top of the stack. Correspondingly, a state of a Boolean Program is a location in the program and a valuation to the variables in scope. An efficient algorithm to decide the reachablity of configurations in pushdown systems is presented in [EHRS00].

6.2 Translating Boolean Programs to Pushdown Systems

In the following, we give a brief overview of how Boolean Programs can be translated to pushdown systems. A more detailed presentation of these matters can be found in [ES01].

First we represent each procedure of the boolean program as *flow graph*. The edges of these flow graphs are associated to the statements of the Boolean Program, whereas the nodes correspond to control points in the procedure.

Symbols of the stack alphabet are used to encode valuations to local variables of procedures and program points. A stack symbol is a tuple $(n, l) \in \Gamma$, where n represents a point in the Boolean Program (respectively a node of one of the above mentioned flow graphs) and l represents a valuation to the local Boolean variables that are in scope at this program point. Such a valuation can be represented as binary string, i.e. $L = \{0,1\}^k$ for k local variables. The size of the stack alphabet depends on the largest number of local variables in any procedure, since local variables of two different procedures cannot be in scope at the same time. The global variables are encoded in the control locations, i.e. if the program contains m global Boolean variables, then they can be encoded as binary strings $G = \{0,1\}^m$.

Each program statement can be represented as transition in the pushdown system. These transitions can be derived from the flow graphs of a boolean program. We explain how this can be accomplished for assignments, procedure calls and return statements:

- *Assignments.* An assignment potentially changes the global or local variables. Therefore, we translate an assignment to a set of rules of the form

  $$\langle g, (n_1, l)\rangle \hookrightarrow \langle g', (n_2, l')\rangle$$

  where the control location g and g' correspond to the valuation of the global variables before and after the assignment, and l and l' represent the local variables before and after the assignment. n_1 is corresponds to the node in the control flow graph that comes before the assignment statement, and n_2 corresponds to the node after the assignment statement.

- *Procedure calls.* A procedure call in a Boolean Program corresponds to *pushing* the local variables of the calling procedure $\mathcal{S}$ and the return address on the stack and transferring the control flow to the entry point of the called procedure $\mathcal{R}$. This is represented by a transition rule

  $$\langle g, (n_1, l_\mathcal{S})\rangle \hookrightarrow \langle g, (m_0, l_\mathcal{R})(n_2, l_\mathcal{S})\rangle$$

  where m_0 denotes the entry point of the procedure $\mathcal{R}$, n_2 represents the return location, and $l_\mathcal{R}$ denotes the initial values of the local variables of $\mathcal{R}$. The parameters are represented as local variables of the procedure, i.e. the procedure has several entry points with different values of $l_\mathcal{R}$, one for each valuation of the parameter predicates.

- *Return statements.* Returning from a procedure call means popping the local variables from the stack and transferring the control flow to the return location. Therefore, the transition rule for a return statement has an empty right side:

  $$\langle g, (n, l)\rangle \hookrightarrow \langle g', \epsilon)\rangle$$

  The global variables may change, since procedures which return values can be simulated assigning the return values to additional global variables.

Example 6.1. *Figure 6.1 shows the flow graph for the procedure* `testdev_open` *presented in Figure 3.4(b) and parts of the flow graph for the code in Figure 3.5 (the* `main` *function)*[1]*. In the*

[1] Notice that the call to the `choose` function is handled as single assignment. Internally, `g := choose(e,f)` is translated to the Boolean expression $(e \wedge g') \vee (\neg e \wedge f \wedge \neg g') \vee (\neg e \wedge \neg f)$, where g' denotes the value of g after the assignment.

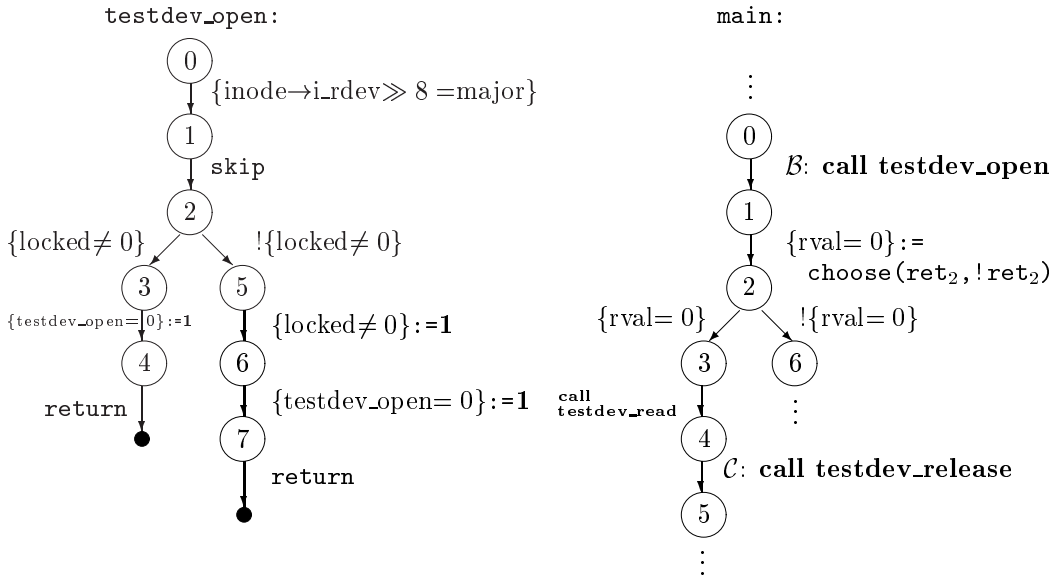


Figure 6.1: Representing Boolean Procedures as Flow Graphs

function **main**, *between control location 0 and 1 a call to* **testdev_open** *is performed. If* l_{main} *represents the local variables of* **main** *at the time of the function call, then the corresponding transition function is*

$$\langle p_0, (main_0, l_{main})\rangle \hookrightarrow \langle p_0, (testdev\_open_0, l_{testdev\_open})(main_1, l_{main})\rangle$$

where $testdev\_open_0$ *represents the entry point of the procedure and* $l_{testdev\_open}$ *represents the initial values of the local variables of the called procedure. The values of the parameters are encoded in* $l_{testdev\_open}$.

The conditional statement in function **testdev_open** *is represented by two transition relations (starting at different control locations, because the predicate* $\{locked\neq 0\}$ *is a global variable:*

$$\langle p_0, (testdev\_open_2, l_{testdev\_open})\rangle \hookrightarrow \langle p_0, (testdev\_open_3, l_{testdev\_open})\rangle$$
$$\langle p_1, (testdev\_open_2, l_{testdev\_open})\rangle \hookrightarrow \langle p_1, (testdev\_open_5, l_{testdev\_open})\rangle$$

Obviously, the conditional statement does not change the state.

The assignment statement $\{locked\neq 0\}$**:=1** *changes the global variables, while* $\{testdev\_open= 0\}$**:=1** *changes the local variables:*

$$\langle p_1, (testdev\_open_5, l_{testdev\_open})\rangle \hookrightarrow \langle p_2, (testdev\_open_6, l_{testdev\_open})\rangle$$
$$\langle p_2, (testdev\_open_6, l_{testdev\_open})\rangle \hookrightarrow \langle p_2, (testdev\_open_7, l'_{testdev\_open})\rangle$$

Finally, the return statement changes the global variables, because the return value is simulated by a global variable:

$$\langle p_2, (testdev\_open_7, l'_{testdev\_open})\rangle \hookrightarrow \langle p_3, \epsilon\rangle$$

The call to the return statement uncovers the return location and the valuation of local variables before the call, which would be $(main_1, l_{main})$ *in our case.*

Reachability in pushdown systems is decidable, and an efficient algorithm is presented in [EHRS00]. This algorithm is implemented in the MOPED model checker. The basic idea behind the algorithm is, that in a pushdown system, the elements relevant for the control flow are the actual control state and the symbol at the top of the stack. There are only finitely many such tuples $\langle p, \gamma \rangle$, since the set of control states as well as the number of stack symbols is finite. If a configuration is encountered repeatedly (with increasing stack size) then the model checker can conclude that the investigated path is looping and further investigation will not lead to new configurations. Therefore, the model checker is able to calculate all reachable configurations without getting caught in a loop.

In all other chapters, we treat MOPED as a black box, i.e. an oracle that decides if a label is reachable in a given Boolean Program. If the label is reachable, MOPED provides a detailed description of how it can be reached (in terms of program points and valuations to the predicates). We map this error trace back to the original C program and check the *feasability* of the resulting path. The next chapter explains how such a feasability check can be accomplished.

Chapter 7

Feasibility of Paths and Refinement

This chapter discusses the algorithm that we use to discover predicates for iterating the abstraction algorithm. We will compare our approach to the algorithm used by the SLAM toolkit. Finally we will show that the whole abstraction/refinement algorithm is potentially non-terminating.

In Chapter 4 we have presented an abstraction algorithm that simplifies a C program to such an extent that a model checker can be applied to check for the reachability of a certain label in this program. We have argued that an abstraction is valid when it allows at least all paths that are feasible in the original program. This means that the set of paths that is feasible in the abstract program is a superset of the set of paths that is feasible in the original C program. Therefore, an error path reported by the model checker does not always imply an error in the C program.

Example 7.1. *In Chapter 3 (Figure 3.4(a)) we have presented the second abstraction of the function* `testdev_open` *(for the original C function refer to Figure 3.2). The model checker reported an error path that calls* `testdev_open` *twice and chose the* then*-branch of the conditional statement each time. This error path is spurious, since it neglects the fact that a lock is maintained to prevent simultaneous access to the device.*

We have to inspect the path in the original C program and check if it is feasible. If this is not the case, we have to refine the abstraction to such an extent that the spurious error path is eliminated.

This chapter covers the algorithm that we use to check the feasibility of a path and to find new predicates for a refining the abstraction. Since our algorithm differs from the approach that the SLAM toolkit uses, we will also give a brief introduction to the algorithm presented in [BR02] and compare it to our solution.

7.1 Feasibility of non-branching Paths

Paths reported by the model checker do not contain any branching statements. If we map the Boolean path back to the corresponding concrete path in the C program, we get a path that contains assignment statements, function calls, scope statements and assumptions. Function calls are expanded by replacing them by a basic block that contains assignments of the actual

```
   locked = FALSE;          // in init_module ()
   ...
A: {                        // call to testdev_open ()
      struct inode   *tmp1;
      struct file    *tmp2;
      tmp1 = &inode;
      tmp2 = &my_file;
      inode = tmp1;
      filp = tmp2;
      assume ((inode->i_rdev) >> 8 ==  major);
      usecount = usecount + 1;
      assume (locked);
   }
```

Figure 7.1: Example for an expanded linear path

s ≡ `x = e`	...	$\mathcal{P} = \{p \mid p = \mathcal{WP}(s, q) \wedge q \in \mathcal{Q}\}$
s ≡ `assume`(φ)	...	$\mathcal{P} = \mathcal{Q} \cup \{\varphi\}$
declaration of $\mathcal{V} = \{\text{var}_1, \text{var}_2, \ldots, \text{var}_n\}$	...	$\mathcal{P} = \mathcal{Q} \setminus \{p \in \mathcal{Q} \mid vars(p) \cap \mathcal{V} \neq \emptyset\}$
end of scope	...	$\mathcal{P} = \mathcal{Q}$

Figure 7.2: Calculating pre-conditions for linear paths

parameters to the formal parameters[1] followed by the function body.

Example 7.2. *We consider a sub path of the path we discussed in Example 7.1. The code in Figure 3.3 calls* `init_module` *and initializes the variable* `locked` *to* `FALSE`*. At label* $\mathcal{A}$ *the function* `testdev_open` *is called. Within the function body the* then*-branch of the conditional statement is executed. The corresponding expanded linear path is shown by Figure 7.1.*

The feasibility of such expanded paths can be checked by calculating the weakest preconditions for the contained statements (see Section 4.3 and Definition 2.8 in Chapter 2). Each assume statement introduces a condition that must hold at the corresponding program point. Showing that a path is infeasible means proving that one of these conditions does not hold.

Each element of the expanded path has a set of pre-conditions $\mathcal{P}$ and a set of post-conditions $\mathcal{Q}$. The post-conditions of an element s_i coincide with the pre-conditions of the subsequent element s_{i+1}. A path is infeasible if we can find a program point i where the conjunction of the conditions in $\mathcal{P}_i$ yields a contradiction.

For an assignment statement s we can calculate the set of pre-conditions from the set of post-conditions by applying $\mathcal{WP}(s, \varphi)$ on each $\varphi \in \mathcal{P}$. An assumption φ yields a set of pre-conditions that contains φ and all post-conditions. The set of pre-conditions $\mathcal{P}$ of a *beginning-of-scope* statement contains all elements of the set of post-conditions $\mathcal{Q}$, except those, that contain variables declared in the new scope. An *end-of-scope* scope statement has no effect on the conditions. The different cases are summarized in Figure 7.2.

[1] Due to recursion, the set of variables in the actual parameters may overlap with the set of formal parameters. We bypass this problem by introducing intermediate variables with fresh names in the assignment block.

	Statement	Conditions
ℓ_0:	`locked = FALSE;`	{FALSE}, {&inode->i_rdev>>8==major}
ℓ_1:	`{ struct inode* ...`	{locked}, {&inode->i_rdev>>8==major}
ℓ_2:	...	{locked}, {&inode->i_rdev>>8==major}
ℓ_3:	`assume ((inode->i_rdev)>>8==` major);	{locked}, {inode->i_rdev>>8==major}
ℓ_4:	usecount = usecount + 1;	{locked}
ℓ_5:	`assume (locked);`	{locked}
		$\emptyset$

Figure 7.3: Showing the infeasibility of the path in Figure 7.1

The calculation of the pre/post-conditions starts with an empty set of post-conditions at the label that the model checker reports to be reachable and is performed from the end of this path towards the beginning of the path. The calculation is stopped if either the beginning of the path is reached or if the conjunction of the pre-conditions at the actual program point yields a contradiction. In the former case, the path is feasible and is reported to the user. In the latter case, the path is infeasible and the abstraction must be refined.

Example 7.3. *Figure 7.3 shows corresponding pre-conditions (resp. post-conditions) to the statements in the path presented in Figure 7.1. For a better understanding the statements in Figure 7.3 should be considered from bottom to top.*

We start at the end of the path with an empty set of post-conditions. The `assume (locked);` *statement at label ℓ_5 adds a new predicate to the set of pre-conditions. Since this condition is not contradictory we have to continue our inspection. The assignment at label ℓ_4 leaves our set of conditions unchanged, while the assumption at ℓ_3 enlarges it by one predicate. Still, the conjunction of the two predicates yields no contradiction. The effect of the following assignment statements (indicated by the ellipsis) is a substitution of* `&inode` *for* `inode` *in the predicate {inode->i_rdev>>8==major}. Note that basic blocks are processed in one step, i.e. the intermediate results for each single assignment statement are not considered (this prevents predicates that reference temporary variables that were introduced for function calls). The scope definition at ℓ_1 does not define any variables that are mentioned by the actual predicates, therefore, the predicates stay unmodified. Note that the parameters are local to the called function* `testdev_open`*, too. But the local parameter* `inode` *refers to a different location than the variable* `inode` *in the predicate {&inode->i_rdev>>8==major} (the latter refers to the location that was handed over as actual parameter). Therefore we keep the predicate. Finally, at ℓ_0, $\mathcal{WP}$("locked =* `FALSE'`*",{locked}) yields the pre-condition {*`FALSE`*}, which is definitely a contradictory predicate. Therefore, the concrete path is infeasible.*

7.2 Discovering Refinement Predicates

The infeasibility of a path reported by the model checker reveals the necessity to refine the abstract program. As we have already stated above, we must additionally consider a set of predicates that makes the spurious error path infeasible. By definition of the weakest precondition (Definition 2.8) the set of all predicates encountered during the feasibility computation suffices to explain the infeasibility. If we consider Example 7.3, an abstraction over the predicates {FALSE}, {locked}, {inode->i_rdev>>8==major}, and {&inode->i_rdev>>8==major} together with the predicates that were used to generate the abstract program in Figure 3.4(a)

would provide enough detail to make the spurious error path infeasible. Therefore, for an infeasible path $s_1, \ldots, s_n$ with the corresponding sets of preconditions $\mathcal{P}_1, \ldots, \mathcal{P}_n$ the predicate set

$$\bigcup_{i=1,\ldots,n} \mathcal{P}_i \tag{7.1}$$

yields an abstraction that does not contain this very path.

A first implementation of this algorithm showed immediately that it yields far too many predicates. In the example from above, the predicate {`FALSE`} is of absolutely no use for the abstraction algorithm, since it does not add any information. Even for simple examples, the calculation of the abstract programs became unacceptably long after a few iterations. Therefore, we implemented a more sophisticated algorithm that reduces the number of predicates that explain the infeasibility of a spurious error path.

We introduce the notion of a *condition thread*. Each predicate in the set of weakest preconditions $\mathcal{P}_i$ for a statement s_i originates from an assumption statement s_j with $i \leq j$. It was either directly introduced by the assumption statement s_j or it resulted from a (repetitive) application of the predicate transformer $\mathcal{WP}$ on the predicate that was introduced by the assumption statement s_j. We denote such a predicate as $p_{i,j}$ and define it as follows:

Definition 7.1. *Let s_j be the assumption* `assume`(φ). *If s_i is not a scope statement and $i < j$, then*

$$p_{i,j} = \begin{cases} \mathcal{WP}(s_i, p_{i-1,j}) & \text{if } s_i \text{ is an assignment statement} \\ p_{i-1,j} & \text{if } s_i \text{ is an assumption} \end{cases} \tag{7.2}$$

If s_i is a beginning-of-scope *statement that contains a declaration of one of the variables referenced in $p_{i-1,j}$, then $p_{i,j}$ is undefined. If s_i is a* beginning-of-scope *statement that does not declare any variables referenced by $p_{i-1,j}$, then $p_{i,j} = p_{i-1,j}$. If s_i is an* end-of-scope *statement, then $p_{i,j} = p_{i-1,j}$. If $i = j$, then $p_{i,j} = \varphi$. For $i > j$, $p_{i,j}$ is not defined.*

Definition 7.2 (Condition Threads). *A* condition thread *is a sequence of predicates $[p_{i,j}, p_{i-1,j}, \ldots, p_{j,j}]$ corresponding to a (sub-)path $s_i, \ldots, s_j$, where s_j is an assumption statement. We denote the first element of this list the* head *of the condition thread.*

Corollary 7.3. *If the head of a condition thread is a predicate representing a false formula then the corresponding (sub-)path is infeasible.*

A path may have more than one associated condition thread. Each assumption introduces a new condition thread. Each *beginning-of-scope*-statement potentially *kills* a condition thread (i.e., the predicates that occur in this condition thread have no influence on the feasibility of the path anymore, because the head of this condition thread is undefined).

As in the naïve approach, we process the path from its end towards its beginning. The condition threads are constructed on the fly. In each step we consider the conjunction of the heads of the actual condition threads (this corresponds to the conjunction of the preconditions for the actual statement, as described in Section 7.1). The path is infeasible iff this conjunction yields a contradiction (due to the definition of the weakest precondition).

Example 7.4. *Figure 7.4 shows condition threads associated with a the path of Figure 7.1. The* `assume`$(locked)$ *statement introduces a new condition thread. After processing the assignment statement* `usecount = usecount + 1;` *the head of this condition thread is still* $\{locked\}$,

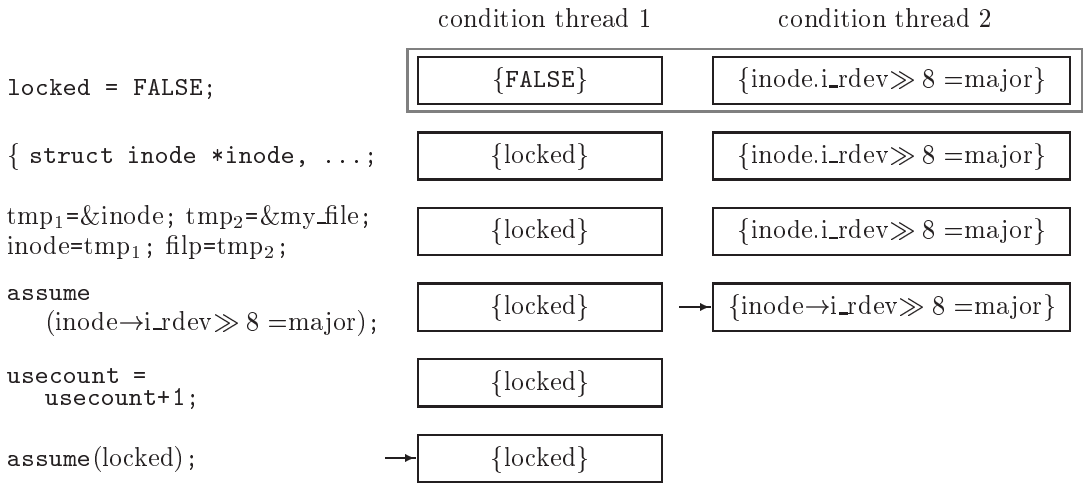


Figure 7.4: Condition Threads

since the weakest precondition for this assignment leaves the predicate unmodified. The next assumption statement introduces a second condition thread, but leaves the head of the first condition thread unmodified. The basic block modifies the heads of the second thread. The declarations of the beginning-of-scope-*statement do not contain variables that are referenced by the head of either thread, therefore none of the two threads is killed. Finally, the head of the first thread is modified by the predicate transformer $\mathcal{WP}$ for* `locked = FALSE;`.

In each step, we check if the conjunction of the heads of the actual threads yields a contradiction. If this is the case, the computation can be terminated and the infeasibility of the path is proved. If the complete path has been processed and the conjunction of the heads is no false formula then the path is feasible.

If the inspected path turns out to be infeasible we have to determine a set of predicates that explains the infeasibility. If one of the heads of the condition threads represents a false formula then the elements of this thread are sufficient to explain the infeasibility (due to Corollary 7.3). Otherwise, we try to determine a subset of condition threads that suffices to explain the infeasibility.

We consider the *shortest* thread (which is always the one that has been introduced last) first and work towards the *longest* thread (the one that has been introduced first). We do not consider killed threads. The computation involves the following steps:

1. Sort the heads p_{k,j_1}, p_{k,j_2}, ..., p_{k,j_n} by ascending order of j_i (i.e., consider the shortest threads first) and store them in a list. Start with an empty set of predicates and call it Γ.

2. Consider the next head in the list and add it to Γ. Construct the conjunction $\bigwedge_{\gamma_i \in \Gamma} \gamma_i$ and check if it yields a contradiction. If this is the case, continue with step 3, otherwise reiterate step 2.

3. The element p_{k,j_i} that was added to Γ last is decisive for the contradiction. Therefore, compute the syntactic cone-of-influence as introduced in Section 4.4.5 by Formula 4.10 and remove all predicates from Γ that are not involved in the contradiction.

4. The threads whose heads are in Γ are sufficient to explain the infeasibility of the path. Therefore, add the elements of these threads to Γ.

5. Finally, remove all predicates that are false from Γ. Return Γ.

Note that this algorithm does not yield a minimal set of predicates that explain the feasibility. The result could be improved by eliminating one predicate at a time and by checking if the remaining set of predicates is still sufficient to explain the infeasibility. However, we have found that in practice the resulting set of predicates is a good approximation.

Example 7.5. *The algorithm discovers that the predicate* {locked} *is sufficient to explain the infeasibility of the path presented in Figure 7.1. This is because the head of the first condition thread in Figure 7.4 is* `FALSE`*, and therefore the predicates of the second thread are discarded. Considering this predicate in the next abstraction yields a new abstract program that does not allow this very path to be feasible.*

The watchful reader will have noticed that we have ignored the problems that occur in connection with pointers and aliases in a path. The reason is that our current implementation of BOOP does not support pointers, as we have already mentioned in Section 4.3.1. However, since the path does not contain any branching statements, it would be possible to determine the exact location that is aliased by a pointer with help of a pointer analysis.

7.3 Comparison to Newton

As we have already mentioned, our feasibility checking algorithm differs from the algorithm used by the SLAM Newton tool. We give a short summary of the algorithm presented in [BR02] and compare its results to the results of our algorithm.

Instead of using the $\mathcal{WP}$ predicate transformer, Newton calculates the *strongest post-condition* of the given path. In [BR02] the strongest post-condition (SP) is defined as follows:

$$\begin{aligned} SP(x := e) &= \lambda f.\exists x'.f[x'/x] \wedge (x = e[x'/x]) \\ SP(\texttt{assume}(e)) &= \lambda f.f \wedge e \end{aligned} \tag{7.3}$$

For a sequence of statements, $p = s_1, s_2, \ldots, s_n$, $SP(p) = SP(s_n) \circ SP(s_{n-1}) \circ \ldots \circ SP(s_1)$, where $g \circ h$ denotes the functional composition $\lambda x.g(h(x))$ of two functions. The quantifiers are eliminated by introducing *Skolem constants*. The path is processed from its beginning towards its end, and each time when a variable x is used in statement s without being defined, a "symbolic constant" θ_x is introduced and "$x := \theta_x$" is inserted immediately before statement s.

Newton uses a a slightly modified version of the strongest post-condition to *symbolically simulate* the path. For this purpose, Newton maintains the following three components:

- Ω maps each variable to its actual "value", i.e. if an assignment $x := e$ associates a variable with an expression e, then the updated Ω will map x to $\Omega(e)$.

- Φ is used to store the *conditions* introduced by assumptions. Each time an assumption `assume`(e) is encountered, $\Omega(e)$ will be added to the set of conditions.

- Π is a relation that represents the past valuations to the variables in the path. If an assignment statement $x := e$ is encountered, then $\langle x, \Omega(x)\rangle$ is added to Π before Ω is updated (if x is an element of the domain of Ω).

NEWTON processes the path p until it either its end is reached or $SP(p')(\mathbf{true})$ becomes false for a sub-path p' of p. In the former case, the path p is feasible. In the latter case, p' is infeasible. Then the set of predicates that explains the infeasibility is extracted as follows:

For each pair $\langle x, e\rangle$ in Π let $\mathcal{C}(\langle x, e\rangle)$ denote the boolean expression $(x =_s e)$ (where $=_s$ is used to distinguish this equality from equalities that arise from `assume` statements). $\mathcal{C}$ is defined similar for $x \mapsto e$ in Ω. $\mathcal{C}$ is generalized to maps and sets of pairs the usual way. Then, the set of predicates $E = C(\Omega) \cup \mathcal{C}(\Pi) \cup \Phi$ is sufficient to explain the infeasibility of p.

NEWTON attempts to reduce the number of these predicates by considering sub-paths of p and by applying a heuristic to eliminate predicates.

NEWTON attempts to get rid of the Skolem constants by post-processing the resulting predicates. The post-processing step keeps θ_x if x has been updated since the initial use of x and eliminates θ_x otherwise. NEWTON is not always able to replace the Skolem constants. Therefore, the predicates that NEWTON discovers may contain Skolem constants that do not occur in the original C program. Therefore, C2BP, the abstraction tool of the SLAM toolkit, has to insert additional assignments into the code before a new abstraction can be calculated.

Since BOOP uses the weakest pre-condition instead of the strongest post-condition to check the feasibility of a path, it is not necessary to introduce Skolem constants. Therefore, there is also no necessity to modify the original C program.

The advantage of the algorithm that NEWTON uses is that it augments Φ resp. Π with only one predicate for each processed statement. BOOP adds one predicate if an assumption is encountered, but it potentially adds one new predicate to *each* condition thread if an assignment is encountered. However, BOOP implicitly performs a data flow analysis [ASU86], i.e. assignments that have no influence on predicates that occur later in the program are ignored. Furthermore, if a predicate is not affected by an assignment, no new predicate is added.

Example 7.6. *Figure 7.5 shows the intermediate results*[2] *of the feasibility calculation of* NEWTON *for the path presented in Figure 7.1. Obviously, these intermediate results involve a number of predicates that are not necessary to explain the infeasibility. We assume that* NEWTON *can discard most of these predicates with help of the heuristic that is mentioned (but not explained in detail) in [BR02]. However, if* NEWTON *fails to remove all predicates that refer to the Skolem constant* θ_3*,* C2BP *has to insert the assignment "usecount= θ_3" into the original C program.*

7.4 Problematic C Programs

We have already stated that there are C programs that force the SLAM and BOOP toolkits to iterate very often. In fact, due to the undecidability of the halting problem, there are also

[2] NEWTON handles pointers and aliasing slightly different than presented here. However, for reasons of simplicity we omit a detailed description of these matters.

Path $p1$	Ω	Φ	Π
`locked = FALSE;`	locked$\mapsto$`FALSE`		
&inode $= \theta_1$	locked$\mapsto$`FALSE`, &inode$\mapsto \theta_1$		
tmp_1 = &inode;	locked$\mapsto$`FALSE`, &inode$\mapsto \theta_1$, $\text{tmp}_1 \mapsto \theta_1$		
&my_file $= \theta_2$	locked$\mapsto$`FALSE`, &inode$\mapsto \theta_1$, $\text{tmp}_1 \mapsto \theta_1$, &my_file$\mapsto \theta_2$		
tmp_2 = &my_file;	locked$\mapsto$`FALSE`, &inode$\mapsto \theta_1$, $\text{tmp}_1 \mapsto \theta_1$, &my_file$\mapsto \theta_2$, $\text{tmp}_2 \mapsto \theta_2$		
inode = tmp_1;	locked$\mapsto$`FALSE`, &inode$\mapsto \theta_1$, $\text{tmp}_1 \mapsto \theta_1$, &my_file$\mapsto \theta_2$, $\text{tmp}_2 \mapsto \theta_2$, inode$\mapsto \theta_1$		
...	...		
usecount = θ_3	locked$\mapsto$`FALSE`, &inode$\mapsto \theta_1$, $\text{tmp}_1 \mapsto \theta_1$, &my_file$\mapsto \theta_2$, $\text{tmp}_2 \mapsto \theta_2$, inode$\mapsto \theta_1$, usecount$\mapsto \theta_3$, ...	...	
usecount = usecount + 1;	locked$\mapsto$`FALSE`, &inode$\mapsto \theta_1$, $\text{tmp}_1 \mapsto \theta_1$, &my_file$\mapsto \theta_2$, $\text{tmp}_2 \mapsto \theta_2$, inode$\mapsto \theta_1$, usecount$\mapsto \theta_3 + 1$, ...	...	$\langle$usecount,$\theta_3\rangle$
`assume` (locked);	locked$\mapsto$`FALSE`, &inode$\mapsto \theta_1$, $\text{tmp}_1 \mapsto \theta_1$, &my_file$\mapsto \theta_2$, $\text{tmp}_2 \mapsto \theta_2$, inode$\mapsto \theta_1$, usecount$\mapsto \theta_3 + 1$, ...	(`FALSE`),...	$\langle$ usecount,$\theta_3\rangle$

Figure 7.5: NEWTON Feasibility Calculation for Figure 7.1

programs for which the algorithm does not converge[3]. A high number of iterations results in an unacceptable response time of the toolkit. In this section, we present a class of programs for which the approach is practically inapplicable.

Remember that BOOP is forced to reiterate if an error path that the model checker suggests is infeasible. Assume that the label ℓ in a C program is only reachable if a loop is iterated $\mathcal{K}$ times:

```
i = 0;
while (i < K) {
   ...
   i = i + 1;
}
ℓ:
```

It is necessary that ℓ is indeed reachable and that there are no other assumptions that

[3]In [BR01] it is stated that the SLAM refinement algorithm may not converge due to the undecidability of property checking

make the path infeasible. Let us inspect the first few iteration steps our algorithm would pass through for the given C program:

1. Assume that the abstraction algorithm is started with the empty set of predicates. Obviously, the model checker will find a path that reaches the label ℓ. However, the refinement algorithm cannot prove the feasibility of such a path, since for this purpose it would have to show that the guard of the loop is false at label ℓ. This attempt must fail, since $(1 < \mathcal{K})$ evaluates to true under the assumption that $\mathcal{K}$ is very large.

2. The refinement algorithm suggests the new predicate $\{\neg(i < \mathcal{K})\}$ to improve the abstraction. Thus, the abstraction algorithm will generate a boolean program similar to the following program:

   $\{\neg(i < \mathcal{K})\}$:= choose(**0**,**1**);
   while (*) do
      assume(!$\{\neg(i < \mathcal{K})\}$);
      ...
      $\{\neg(i < \mathcal{K})\}$:= choose ($\{\neg(i < \mathcal{K})\}$, **0**);
   od
   assume($\{\neg(i < \mathcal{K})\}$);
   ℓ:

3. Again, the model checker will find a path that reaches the label ℓ by executing the loop body once, because choose($\{\neg(i < \mathcal{K})\}$, **0**) returns **0** or **1** non-deterministically (note that $\{\neg(i < \mathcal{K})\}$ is **0** when choose is called).

4. Of course, the path found by the model checker is not feasible, since $\neg(i < \mathcal{K})$ cannot be true after one iteration of the loop. By investigating the spurious counter example, the refinement algorithm will find one new predicate, namely $\neg(i + 1 < \mathcal{K})$.

5. After several iterations of the abstraction/refinement algorithm, the boolean program will contain a block of assignments that represents a binary counter:

   $\{\neg(i < \mathcal{K})\}$,
   $\{\neg(i + 1 < \mathcal{K})\}$,
   $\{\neg(i + 1 + 1 < \mathcal{K})\}$:=
      choose ($\{\neg(i < \mathcal{K})\} \vee \{\neg(i + 1 < \mathcal{K})\}$, !$\{\neg(i + 1 + 1 < \mathcal{K})\} \vee$!$\{\neg(i + 1 < \mathcal{K})\}$);
      choose ($\{\neg(i < \mathcal{K})\} \vee \{\neg(i + 1 < \mathcal{K})\} \vee \{\neg(i + 1 + 1 < \mathcal{K})\}$, !$\{\neg(i + 1 + 1 < \mathcal{K})\}$),
      choose ($\{\neg(i < \mathcal{K})\} \vee \{\neg(i + 1 < \mathcal{K})\} \vee \{\neg(i + 1 + 1 < \mathcal{K})\}$, **0**);

   After each iteration, the feasibility checking algorithm adds a new predicate to the abstraction predicate set. This is crucial, since it prevents us from aborting the abstraction/refinement cycle[4]. Only after $\mathcal{K} + 1$ iterations BOOP determines that the path is feasible (because $\neg(i + \mathcal{K} < \mathcal{K})$ is true for $i = 0$).

[4] If a greatest fixpoint of the abstraction predicate set is found, then execution can be aborted, since an additional iteration would yield no new results

A similar problems occurs for programs that use recursion instead of iteration. Therefore, the algorithm is inadequate for such programs. A typical example would be a program that contains an off by one error or a buffer overflow error.

Chapter 8

Conclusions

In this chapter we summarize the work presented in this thesis and present an assessment of the results. Finally we point out open problems and gives suggestions for future work.

8.1 Synopsis

In this thesis we have presented an approach for automatic software verification based on the SLAM project [BR01] of Microsoft Research. We have described how C programs can be abstracted to such an extent that a model checking tool can be applied to determine the reachability of erroneous states. We used predicate abstraction to translate C programs to Boolean Programs. The concrete states of the C program were mapped to abstract states, which were represented by a finite set of predicates. We presented the algorithm that we used to calculate the abstract state transitions that approximate the changes of the concrete state space. Furthermore we proved the correctness of this algorithm.

We presented a formalization of C expressions in terms of the logic of the HOL98 theorem prover.

Furthermore we explained how the model checker MOPED translates Boolean Programs to pushdown systems. We discussed why the counter examples provided by MOPED may present spurious execution traces that are not feasible in the original C program, and we presented a way to determine the feasibility of such paths. Furthermore we explained how such spurious counter examples can be used to refine the abstraction. Our algorithm for discovering new predicates differs from the algorithm implemented in the NEWTON tool, and we have pointed out its advantages and disadvantages.

We have provided an example of how the approach can be used to validate temporal safety properties, and we showed that the approach is not suitable for a certain class of programs.

8.2 Assessment of Results

Due to the early state of the implementation, the BOOP toolkit could only be applied to relatively small examples. Based on the results of the tests we carried out, we draw the following conclusions:

- *Applicability.* We believe that the approach is suitable for industrial development, since it requires no scientific skills of the programmer. Since there is no need to learn a

specification language (and therefore no need for initial training) the method is likely to be accepted by programmers. The fact that the approach was developed by scientists at Microsoft Research raises the hope that it will soon be integrated into widely-used development environments.

- *Scalability.* We were not able to perform a practical evaluation of the scalability of our approach, because the BOOP toolkit is not yet applicable to real world C programs. In Section 7.4 we have presented a class of programs that demand an unacceptably large number of abstraction/refinement cycles, which make the algorithm practically infeasible. Furthermore, software systems that make heavy use of complicated (floating point) computations are likely to overcharge the theorem prover.

  The applicability for validation of temporal safety properties in operating systems is not affected by these restrictions. We believe that the approach is well suited for the verification of device drivers and embedded systems software, due to the lack of floating point computations in such applications. Experimental results of other research teams [BR01, HJMS02] confirm this assumption.

8.3 Open Problems and Future Work

There are several unfinished tasks and open problems that could not be addressed within the scope of this thesis. First, we summarize the problems that can be easily overcome by investing more resources. Then we discuss several inherent limitations of the approach.

We provide an enumeration of problems that need to be addressed by future projects. We do believe that no further theoretical investigation is necessary to solve these problems; they are mainly of practical nature.

- *Pointer Analysis.* Due to the lack of a pointer analysis, the current implementation of BOOP is not applicable to real world C programs like the Linux kernel. The implementation of a pointer analysis algorithm as presented in for example [Ste96] or [Das00] is therefore indispensable in case that the development of the BOOP toolkit is continued.

- *Complete Formalization of C expressions.* The formalization of C expressions (presented in Chapter 5) is incomplete. Several important operations are not yet covered. In particular, it is necessary to formalize bit manipulation operations, due to their heavy use in device drivers.

- *Missing C constructs.* At the moment, not all C constructs are yet covered by BOOP. The most urgent extension would be to handle C expressions with side effects. Further on, several control flow constructs like `for`-loops, `switch`-statements and `case`-labels, `goto`-statements and `break`/`continue`-statements are not yet supported.

- *Repeated Effort in Abstraction Calculation.* At the moment, the abstraction of the complete program is re-calculated in each refinement step, even if parts of the code are not affected by the new predicates. This problem was addressed by the developers of the BLAST toolkit by using *lazy abstraction* [HJMS02]. The BLAST toolkit restricts the code regions that are iteratively abstracted. A similar extension would be desirable for the BOOP toolkit.

- *Optimization of Theorem Proving.* The principal aim of our project was to close the abstraction/refinement cycle. Therefore, our integration of the theorem prover is sub-optimal. Caching of the calls to the theorem prover would be desirable to improve the performance. Furthermore, optimizations of the decision procedure are necessary. It might be worthwhile to investigate the suitability of various formalizations of bounded integer types (e.g. the `word` theory of HOL98, or BDDs) for our approach.

The approach we presented has some inherent restrictions. The following problems are candidates for further theoretical investigation. There might be ways to overcome these problems by extending the approach or by combining it with other approaches.

- *External Functions.* At the moment the programmer must supply stubs for external functions. Partly, this is necessary anyway, because the interface constraints are specified in terms of function stubs. However, the programmer should not be forced to supply stubs for external functions that are not relevant parts of the interface. We are not yet completely sure how (and if) this problem can be overcome, since external functions can potentially change the complete state space. For *open* software systems, i.e. systems that do not use libraries for which source code is not available, no restrictions are imposed. A *specification pool* with stubs for common libraries that are only available as binaries would be an interesting subject for future work.

- *Specifications in Temporary Logic.* The MOPED model checker supports specifications in LTL logic [ES01]. This yields counter examples that do possibly contain iterations (i.e. a path that contains a `while` loop but no conditional statements). We believe that the feasibility of such error traces with cycles is harder to decide than the feasibility of non-cyclic traces; However, we were yet unable to prove this assumption. Extending the specification language with LTL logic would be very desirable.

- *Object Oriented Languages.* Investigation of the extensibility of our approach to object oriented languages would be worthwhile. We believe that the treatment of object oriented languages imposes the need for a more sophisticated pointer analysis. Since C and C++ share the same internal representation in the GCC, extending BOOP to cover C++ should be possible.

Bibliography

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Priciples, Techniques, and Tools*. Addison Wesley, 1986.

[BBJ02] G. S. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, 4th edition, 2002.

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - a second generation of a Java model checker. Workshop on Advances in Verification, 2000.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'01)*, pages 203–213. ACM Press, 2001.

[BMR02] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, Microsoft Corporation, June 2002.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bor00] R. Bornat. Proving pointer programs in hoare logic. In *Mathematics of Program Construction*. Springer Verlag, 2000.

[BR00] T. Ball and S.R. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th international SPIN workshop on Model Checking of Software*, pages 113–130. Springer-Verlag, 2000.

[BR01] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model Checking of Software*, pages 103–122. Springer-Verlag, 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, Microsoft Corporation, January 2002.

[BvW98] R.J. Back and J. von Wright. *Refinement Calculus, A Systematic Introduction*. Springer-Verlag, 1998.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Joehanes, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In

Proceedings of the 22nd International Conference on Software Engineering, June 2000.

[CGP99] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[Das00] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 35–46. ACM Press, 2000.

[DCN+00] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER toolkit. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Verlag, March/April 2000.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. Technical Report TUM-I0002, SFB-Bericht Nr. 342/01/00 A, Technische Universität München, Institut für Informatik, Januar 2000.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., 1972.

[ES01] J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *Proceedings of CAV'01, number 2102 in Lecture Notes in Computer Science*, pages 324–336. Springer-Verlag, 2001.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatsh. für Mathematik und Physik*, 38:173–198, 1931.

[Gor00] M. J. C. Gordon. Reachability programming in HOL98 using BDDs. In *Proceedings of the Conference on Theorem Proving and Higher Order Logic (TPHOL'00)*. University of Cambridge Computer Laboratory, 2000.

[Gri81] D. Gries. *The Science of Programming*. Springer Verlag, 1981.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[Hur01] J. Hurd. Congruence classes with logic variables. *Logic Journal of the IGPL*, 9(1):59–75, January 2001.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall International, 2nd edition edition, 1988.

[Ray98] E.S. Raymond. The cathedral and the bazaar. `http://www.tuxedo.org/~esr/writings/cathedral-bazaar/`, August 1998.

[RC01] A. Rubini and J. Corbet. *Linux Device Drivers.* O'Reilly & Associates, Inc., 2nd edition, 2001.

[Sch86] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[Sch00] U. Schöning. *Logik für Informatiker.* Spektrum Akademischer Verlag, Heidelberg; Berlin, 5th edition, 2000.

[Sch02] K. Schmaranz. *Softwareentwicklung mit C.* Springer-Verlag, 2002.

[Smu92] R. M. Smullyan. *Gödel's Incompleteness Theorems.* Oxford University Press, 1992.

[Sta02] R.M. Stallman. *GNU Compiler Collection Internals (for GCC 3.3).* Free Software Foundation, Inc., January 2002.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 32–41. ACM Press, 1996.

[Uni01a] University of Cambridge. *The HOL System Description (for HOL taupo6)*, February 2001. available at `http://hol.sourceforge.net/`.

[Uni01b] University of Cambridge. *The HOL System Tutorial (for HOL taupo6)*, February 2001. available at `http://hol.sourceforge.net/`.

[Wei01] G. Weißenbacher. Translating a bounded subset of VDM-SL to propositional logic. Unpublished paper, describes a contribution to the VDM-SL Proof Engine implemented as part of the PROSPER project, December 2001.