A Model Checker based on Abstraction Refinement

Diploma Thesis

Andrey Rybalchenko

Saarbrücken
2002

Erklärung

Hiermit erkläre ich an Eides statt, daß ich diese Arbeit selbstständig angefertigt und keine anderen als die angegebenen Quellen verwendet habe.

Saarbrücken, den 3. September 2002

Andrey Rybalchenko

Abstract

Abstraction plays an important role for verification of computer programs. We want to construct the right abstraction automatically. There is a promising approach to do it, called *predicate abstraction*. An insufficiently precise abstraction can be *automatically refined*. There is an automated model checking method described in [2] which combines both techniques, e.g., predicate abstraction and abstraction refinement. The quality of the method is expressed by a completeness property relative to a powerful but unrealistic oracle-guided algorithm.

In this work we want to generalize the results from [2] and introduce new abstraction functions with different precision. We implement the new abstraction functions in a model checker and practically evaluate their effectiveness in verifying various computer programs.

Zusammenfassung

Abstraktion spielt bei der Verifikation von Computerprogrammen eine wichtige Rolle. Unser Wunsch ist es, die richtige Abstraktion automatisch zu finden. Es gibt einen vielversprechenden Ansatz es zu machen, der als *predicate abstraction* bezeichnet wird. Eine Abstraction, die nicht präzise genug ist, kann *automatisch verfeinert* werden. Es gibt eine automatisierte Model Checking Methode, die in [2] beschrieben ist, die beide Techniken, nämlich predicate abstraction und Abstraktionsverfeinerung, miteinander kombiniert. Die Qualität der Methode wird durch eine Vollständigkeitseigenschaft ausgedrückt, die relativ zu einem mächtigen aber nicht realistischen orakel-basierten Algorithmus betrachtet wird.

In dieser Arbeit wollen wir die Ergebnisse von [2] verallgemeinern und neue Abstraktionsfunktionen vorstellen. Außerdem wollen wir die neuen Abstraktionsfunktionen in einem Model Checker implementieren und ihre Effektivität an verschiedenen Beispielprogrammen praktisch evaluieren.

Contents

Chapter 1

Introduction

There is a variety of automated verification methods using different techniques to prove system correctness. All methods use abstraction of the system which properties are verified. Such abstraction is a less complex approximation of the original (concrete) system. The approximation must be safe, i.e., it must contain all possible behaviors of the concrete system.

Most interesting system properties are undecidable, and the number of the system states is in the most cases unmanageably high or even infinite. The undecidability is addressed by providing semi-algorithms which do not terminate on some inputs. Abstraction is used to tackle the complexity.

We want to find the right abstraction automatically. For this purpose we have to come up with an abstraction method which provides an easily computable and sufficiently precise abstraction procedure.

There exists a promising approach to construct abstractions automatically, called *predicate* abstraction. The concrete states of a system are abstracted using a finite set of predicates. An insufficiently precise abstraction can be automatically refined by an abstraction *refinement* procedure. The right abstraction is computed iteratively starting with some initial abstraction and refining it using spurious error traces generated by the model checker. This provides a semi-algorithm that possibly does not terminate on some inputs [13].

An automated model checking method which incorporates predicate abstraction with abstraction refinement is described in [2]. Another issue addressed in that paper is a goodness criterion for a verification method given as a semi-algorithm. Relative completeness property wrt. a powerful but unrealistic (oracle-guided) method is used as a quality measure for the verification method from [2].

In this work we want to generalize the relative completeness result from [2] and introduce new abstraction functions with different precision. We set up a framework which should help to reason about the abstraction functions in a uniform fashion. Furthermore, we want to implement the new abstraction functions in a model checker and practically evaluate their effectiveness.

The thesis is organized as follows:

Chapter 2 provides an introduction into model checking as reachability analysis and framework of abstract interpretation;

Chapter 3 describes the implemented model checking algorithm and its unrealistic oracle-guided counterpart;

Chapter 4 introduces a framework which helps to deal with the generalized setting of the verification method without reasoning by cases for each abstraction function;

Chapter 5 gives a description of the implemented model checker: design, implementation issues, and a short user manual;

Chapter 6 contains results of the experiments with some "real-world" programs (imperative programs, communication protocols, parameterized systems, etc.) using different abstraction functions;

Chapter 7 describes some related model checking methods for software as well as for hardware verification;

Chapter 8 gives a concluding overview and open problems.

Chapter 2

Automated Verification

In this chapter we introduce model checking and abstract interpretation. These two techniques are the key building blocks of our verification method. The major difficulties that arise during the system verification are the following. From the theoretical point of view, most of the interesting system properties are undecidable; from the practical point of view a number of states of a complex program can be unmanageably large or even infinite (known as "state explosion").

2.1 Model Checking

Model checking is an automated technique for verifying (in)finite state systems. It has many advantages over traditional approaches such as simulation, testing, and deductive verification. Among the advantages is the possibility to exhaustively (by using symbolic methods) explore the state space, which can be infinite, to determine whether the system satisfies the given specification.

We restrict the class of problems that our model checker should be able to solve. We are interested in verifying safety specifications, i.e., we want to check whether a particular set of the system states Q_{err} can be reached during some execution of the system. Usually Q_{err} is a set of error states that must not be reachable under assumption that the system is correct. This temporal property is interesting for practical purposes of system verification, and as many other temporal properties, it is undecidable in general.

We assume that the infinite state system that we want to verify can be translated into corresponding transition system $\mathcal{S}$ given as a tuple

$$\mathcal{S} = \langle Q, Q_{init}, \rightarrow \rangle$$

In this tuple Q is a set of states, $Q_{init} \subseteq Q$ is a set of the initial states, and finally $\rightarrow \subseteq Q \times Q$ is a transition relation. The transition relation $\rightarrow$ can be extended to the sets of states $Q_p \subseteq Q$ as follows:

$$\mathsf{post}_{\mathcal{S}}(Q_p) = \{q' \in Q | q \in Q_p \text{ and } (q, q') \in \rightarrow\}$$

The backward iteration operator is defined as:

$$\mathsf{pre}_{\mathcal{S}}(Q_p) = \{q' \in Q | q \in Q_p \text{ and } (q', q) \in \rightarrow\}$$

2.1.1 Reachability Analysis

The reachability analysis can be reduced to the set inclusion test whether the set of the reachable states does not contain Q_{err}.

The set of states (forward-) reachable from Q_{init} is the least solution of the fixpoint equation [11]

$$Reach = Q_{init} \cup \mathsf{post}_{\mathcal{S}}(Reach)$$

We can also reverse the direction of the reachability analysis and test if the set of states backward reachable from Q_{err} entails the set of initial states Q_{init}. The set of backward reachable states can be obtained by finding the least solution of the following fixpoint equation:

$$Backreach = Q_{err} \cup \mathsf{pre}_{\mathcal{S}}(Backreach)$$

The safety property of the system in the sense of the reachability specification can be expressed (depending on the analysis direction) as one of the tests

$$\begin{aligned} Reach \cap Q_{err} &\stackrel{?}{=} \emptyset \\ Backreach \cap Q_{init} &\stackrel{?}{=} \emptyset \end{aligned}$$

The system is safe if the intersection set is empty.

2.2 Abstract Interpretation

The undecidability of the reachability problem and the complexity caused by the state explosion can be tackled by using some form of approximation. The general idea of using approximation is to replace a system by its less complex model. A proof obtained for the model is also a proof for the concrete system.

We can approximate an infinite state system by a finite state model. There are several types of finiteness assumptions: approximation by finite domains, restriction of unbounded data structures, restriction of accepted specification and/or system types, etc.

A unified abstraction framework is needed to set up a model checking method based on approximation. Otherwise, one would finally end in "Teufels Küche".[1]

There is a framework of abstract interpretation introduced in [6, 7] which formalizes the notion of approximation. There are two approaches to abstract interpretation introduced in this framework: a Galois connection approach, and a widening/narrowing approach.

An application of the widening/narrowing approach is a deductive model checker DMC implemented at MPII Saarbrücken [11]. In this work we describe an implementation of a model checker which is based on the Galois connection approach. A brief overview of the approximation methods is given bellow.

2.2.1 Galois Connection Approach

The Galois connection approach to abstract interpretation [7] formalizes the fact that a fixpoint equation $X = F(X)$ can be approximated by some equation $\overline{X} = \overline{F}(\overline{X})$. It is required that $\overline{F} \in \overline{L} \longmapsto \overline{L}$ is a monotone function; $\overline{L}(\overline{\sqsubseteq}, \overline{\sqcup})$ is a finite poset, i.e., a finite set partially ordered by a reflexive, transitive, and antisymmetric order relation $\overline{\sqsubseteq}$. Moreover, $\overline{\sqcup}$ denotes the least upper bound of the poset. $\overline{L}(\overline{\sqsubseteq}, \overline{\sqcup})$ is finite, hence, we can solve the approximating equation $\overline{X} = \overline{F}(\overline{X})$ iteratively starting from the basis $\overline{\bot}$.

Now we need to formalize the correspondence between the concrete domain L and its finite discrete abstraction $\overline{L}$. This is done by a Galois connection (also called *pair of adjoint functions*).

Definition 1 (Galois Connection). *If $L(\sqsubseteq)$ and $\overline{L}(\overline{\sqsubseteq})$ are posets, then $\langle\alpha, \gamma\rangle$ is a Galois connection, written $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{L}$, if $\alpha \in L \longmapsto \overline{L}$ and $\gamma \in$*

[1] German, – *Teufel*, devil, *Küche*, kitchen

$\overline{L} \longmapsto L$ are functions such that:

$$\forall x \in L, \overline{y} \in \overline{L} : (\alpha(x) \overline{\sqsubseteq}\, \overline{y}) \Leftrightarrow (x \sqsubseteq \gamma(\overline{y})) \tag{2.1}$$

$\alpha(x)$ is the abstraction of x, i.e., the most precise approximation of $x \in L$ in $\overline{L}$. $\gamma(\overline{y})$ is the concretization of $\overline{y}$, i.e., the most imprecise element of L which can be soundly approximated by $\overline{y} \in L$.

An example of using a Galois connection can be found in [9] (approximation of sets of natural numbers by intervals).

The abstract domain $\overline{L}$ can be parameterized [2], so it is possible to adjust the approximation precision. For a detailed description of the parameterized Galois connection implemented in the model checker see Section 4.2.3.

2.2.2 Widening/Narrowing Approach

The key idea of the widening/narrowing approach [6, 7] is to use a widening operator $\nabla \in L \times L \longmapsto L$ such that:

$$\begin{aligned} \forall x, y \in L : &\quad x \sqsubseteq x \nabla y \\ \forall x, y \in L : &\quad y \sqsubseteq x \nabla y \end{aligned}$$

and for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ the increasing chain defined by $y_0 = x_0, \ldots, y_{i+1} = y_i \nabla x_{i+1}, \ldots$ is not strictly increasing. If the widening operator is used to iteratively compute the least fixpoint as follows:

$$\begin{aligned} \hat{X}^0 &= \bot \\ \hat{X}^{i+1} &= \begin{cases} \hat{X}^i & \text{if } F(\hat{X}^i) \sqsubseteq \hat{X}^i \\ \hat{X}^i \nabla F(\hat{X}^i) & \text{otherwise} \end{cases} \end{aligned}$$

then the computation terminates and its limit $\hat{A}$ is a sound approximation of the least fixpoint $\mathsf{lfp}(F, \bot)$.

The approximation $\hat{A}$ computed using widening operator can be improved using a narrowing operator $\triangle \in L \times L \longmapsto L$ [9].

Other widening operators (e.g., unary ones) are also possible. E.g., in Chapter 3 we use a unary widening operator $\nabla \in L \longmapsto L$ such that:

$$\forall x \in L : x \sqsubseteq \nabla(x)$$

and the least fixpoint is computed as

$$\begin{aligned} \hat{X}^0 &= \bot \\ \hat{X}^{i+1} &= \nabla \circ F(\hat{X}^i) \end{aligned}$$

An example of using widening/narrowing on intervals can be found in [9].

Chapter 3

Model Checking: Method I and Method II

There is a pair of model checking methods introduced in [2] called Method I and Method II. The first method represents a model checking algorithm which uses predicate abstraction combined with abstraction refinement. The second one is based on fixpoint iteration with unrealistic oracle-guided widening. The algorithms[1] are parameterized by the iteration direction: forward, and backward.

As mentioned in Chapter 2, we try to tackle the undecidability and the complexity of model checking by using some form of approximation. We are interested in approximation techniques that can be automatically adjusted to a particular system. Namely, if the verification algorithm failed to prove the system property, a refined approximation is computed, and the algorithm is given another try.

For our implementation, we are interested in the algorithm instance with backward iteration and backward abstraction refinement. We make the ordering on system states parameterized and take different abstraction functions into consideration.

In the following sections we describe Method I and Method II.

3.1 System Representation

In this section we describe how a system that we want to verify is represented.

[1] These algorithms do not terminate in general so we should call them "semi-algorithms", but we keep the term algorithm for simplicity.

We represent an input system by a set of initial states, error states, and a transition relation. We represent a possibly infinite set of the reachable states symbolically using a formula over the system variables; a system state is a valuation of the system variables. A formula is constructed from a fixed finite set of atomic formulas (we call them *predicates*) using boolean connectives. In our implementation the predicates come from linear arithmetic, but another formalisms can be also used.

The transition relation over the system states is represented as a set of guarded command of the form:

$$\mathtt{Guard}(X) \wedge X' := \mathtt{Update}(X)$$

where X is a tuple of all system variables $\langle X_1, \ldots, X_n \rangle$ including the location label (or program counter). X' denotes the variables after the command execution. The formula $\mathtt{Guard}(X)$ is the enabling condition of the command, i.e., the transition is possible iff the guard is satisfied. The updates of the system variables are defined as $X_i' := \mathtt{Update}_i(X)$ for $i = 1, \ldots, n$, where $\mathtt{Update}_i(X)$ is an arithmetic expression or a new location label.

Here is an example of representing a transition system as a set of guarded commands.

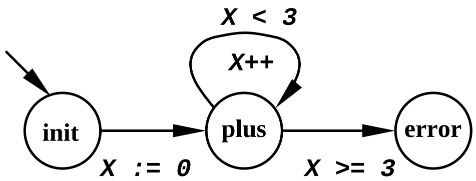

Figure 3.1: A small system.

Example 1 (Guarded Commands). We translate the transition system shown in Figure 3.1 into a set of guarded commands $C = \{c_1, c_2, c_3\}$ given in Table 3.1.

3.2 Iteration and Invariants

Now we describe a "syntactic" transition operator pre, a parameterized order relation on system states $\leq_k$, and formalize the system correctness.

	Guard	Update
c_1	$PC = \texttt{init}$	$PC' := \texttt{plus}, X' := 0$
c_2	$PC = \texttt{plus} \wedge X < 3$	$PC' := \texttt{plus}, X' := X + 1$
c_3	$PC = \texttt{plus} \wedge X \geq 3$	$PC' := \texttt{error}$

Table 3.1: Guarded commands of the small system.

We define the predecessor operator pre_c for every guarded command c of a system. The application of pre_c on a formula φ, with the tuple of free variables X, consists in substituting the variables in X by its primed values, computed using $\texttt{Update}(X)$, and adding the command's guard:

$$\mathsf{pre}_c(\varphi) = \texttt{Guard}(X) \wedge \varphi[\texttt{Update}(X)/X]$$

Note that the execution of pre_c does not require a constraint satisfiability check.

The operator pre for a system $\mathcal{C}$ is a disjunction

$$\mathsf{pre}(\varphi) = \bigvee_{c \in \mathcal{C}} \mathsf{pre}_c(\varphi)$$

The following example shows how the "syntactic" operator pre is executed.

Example 2 (Syntactic Transition). We apply the second command from Example 1

$$c_2 \equiv PC = \texttt{plus} \wedge X < 3 \wedge PC' {:=} \texttt{plus} \wedge X' {:=} X + 1$$

on the formula

$$\varphi \equiv PC = \texttt{plus} \wedge X = 1$$

The tuple of free variables of φ is $\langle PC, X \rangle$. The computation of the transition is done in the following steps:[2]

$$\begin{aligned} \mathsf{pre}_{c_2}(\varphi) \equiv &\quad \texttt{Guard}(\langle PC, X \rangle) \wedge \varphi[\texttt{plus}/PC,\ X + 1/X] \\ \equiv &\quad PC = \texttt{plus} \wedge X < 3 \wedge X + 1 = 1 \\ \equiv &\quad PC = \texttt{plus} \wedge X < 3 \wedge X = 0 \end{aligned}$$

The transition result is the following:

$$PC = \texttt{plus} \wedge X < 3 \wedge X = 0$$

[2]Note that we simplified the substitution of the program counter variable PC in this example.

We want to compare sets of states with each other (whether one set contains more states than other). For this purpose, we introduce an ordering on the sets of concrete states, denoted as $\leq_k$, and an ordering on the sets of abstract states, denoted as $\sqsubseteq$. We use the index k to stress that the concrete ordering is parameterized. For more details see Section 4.2.

In our algorithms, we denote the set of all states except the initial state by the formula $\mathsf{nonInit}$, and the initial set of error states by the formula unsafe. The formula $\mathsf{nonInit}$ is only supposed to express that the location label init is unreachable, i.e., $PC \neq \mathsf{init}$.

We formalize the system correctness defined in Section 2.1.1 as

$$\mathsf{lfp}(\mathsf{pre}, \mathsf{unsafe}) \leq \mathsf{nonInit} \tag{3.1}$$

where $\mathsf{lfp}(\mathsf{pre}, \mathsf{unsafe})$ is the least fixpoint of the operator pre computed from the basis unsafe. Note that we can easily do the entailment check in (3.1) by comparing the location label of the reachable state with the label init. The system is correct if the location labeled by init is not reached.

The system correctness is proved by computing an invariant and proving its safety. The fixpoint $\varphi = \mathsf{lfp}(\mathsf{pre}, \mathsf{unsafe})$ is a safe invariant if the following implications are valid:

$$\begin{aligned} \mathsf{unsafe} &\leq_k \varphi \\ \mathsf{pre}(\varphi) &\leq_k \varphi \\ \varphi &\leq_k \mathsf{nonInit} \end{aligned} \tag{3.2}$$

The correctness can be also established by constructing an upper approximation of the operator pre, computing the least fixpoint of the approximated operator, and proving its safety. The framework of abstract interpretation [6, 7], described in Section 2.2, is used to set up two model checking methods: Method I, and Method II. The first method is based on predicate abstraction; the second one uses widening.

3.3 Method I: Predicate Abstraction with Refinement

Method I is an algorithm for iteratively finding a safe invariant (3.2) of a program or showing the program incorrectness (see Figure 3.2 for main steps of the algorithm). The abstract fixpoint iteration is parameterized by a finite set of predicates $\mathcal{P}_n$, where $n = 0, 1, \ldots$ denotes the number of the refinement step. If the found invariant (i.e., the fixpoint) does not imply the system correctness because the approximation was too coarse then a refined iteration operator is constructed, and the algorithm starts a new fixpoint computation using the refined version of the operator.

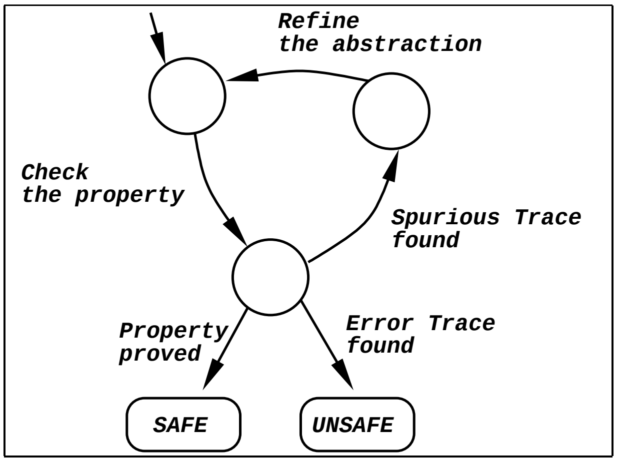

Figure 3.2: Method I: Algorithm with abstraction refinement.

The refinement procedure iterates the concrete operator pre and collects the resulting atomic formulas:

Definition 1 (Operator preds**).**

$$\mathsf{preds}(\bigvee_{i \in I} \bigwedge_{j \in J_i} \varphi_{ij}) = \{\varphi_{ij} | i \in I, j \in J_i\}$$

The set of the atomic formulas grows after each refinement iteration $\mathcal{P}_n = \bigcup_{i=0}^{n} \mathsf{preds}(\mathsf{pre}^i(\mathsf{unsafe}))$; this means that the abstraction quality increases for increasing n.

A detailed description of Method I is shown in pseudo-code notation in Figure 3.3.

The fixpoint φ is computed by iterating the abstract operator $\mathsf{pre}_n^\#$ over the finite free lattice $\mathcal{L}(\mathcal{P}_n, \sqsubseteq)$ which is generated by the set of atomic formulas $\mathcal{P}_n$ and ordered by a lattice ordering $\sqsubseteq$. The abstract transition operator is the most precise abstraction $\mathsf{pre}_n^\#$ of the operator pre wrt. to the parameter set $\mathcal{P}_n$. It is defined as a composition of an abstraction function $\alpha_k^{\mathcal{P}_n}$, the predecessor operator pre and a concretization function

```
P_0 := preds(unsafe)
n := 0
loop
  construct abstract operator pre_n^# defined by P_n
  φ := lfp(pre_n^#, unsafe) /* lfp found after i iterations */
  if (φ ≤_k nonInit) then
      STOP with "Safe"
  else if (pre^i(unsafe) ≰_k nonInit) then
      STOP with "Unsafe"
  P_{n+1} := P_n ∪ preds(pre^{n+1}(unsafe))
  n := n+1
endloop
```

Figure 3.3: Method I: Abstract fixpoint iteration with iterative abstraction refinement.

$\gamma_k^{\mathcal{P}_n}$ using the Galois connection approach to abstract interpretation

$$\mathsf{pre}_n^{\#} = \alpha_k^{\mathcal{P}_n} \circ \mathsf{pre} \circ \gamma_k^{\mathcal{P}_n} \tag{3.3}$$

Note that the abstract operator is parameterized in two ways: by the set of atomic predicates $\mathcal{P}_n$ and by the state ordering $\leq_k$. A detailed description of the operators, order relations, and requirements imposed on them is given in Chapter 4.

3.4 Method II: Oracle-Guided Widening

Method II finds an invariant by iterating the concrete operator pre starting from the set unsafe and querying an oracle to perform widening. The algorithm description is given in Figure 3.4.

The oracle instantly determines a widening operator at each iteration step. The widening operator, provided by the oracle, is applied on the result of the application of pre.

There is a restriction on the widening operator chosen by the oracle: it can only drop some (or none) of the conjuncts of a formula φ. The oracle's answer is a number w which determines the set of conjuncts that are dropped by the widening:

$$\mathsf{widen}(w, \bigvee_{i\in I} \bigwedge_{j\in J_i} \varphi_{ij}) = \bigvee_{i\in I} \bigwedge_{j\in J_i'} \varphi_{ij}, \text{where } J_i' \subseteq J_i \text{ is determined by } w$$

The resulting formula φ' entails φ, i.e., it denotes a larger set of states.

```
φ := unsafe
loop
  φ' := pre(φ)
  if (φ' ≤_k φ) then
     if (φ ≤_k nonInit) then
        STOP with "Safe"
     else
        STOP with "Don't know"
  else
  w := guess provided by oracle
  φ := widen(w, (φ ∨ φ'))
endloop
```

Figure 3.4: Method II: fixpoint iteration with oracle-guided widening.

Chapter 4

Abstraction Refinement Framework

The abstract transition operator $\mathsf{pre}^{\#}$ (3.3) can be constructed using different abstraction functions. Hence, by proving lemmas and theorems about Method I reasoning for every abstraction functions is unavoidable.

We introduce a framework that helps to work with different abstraction functions and to reason about them in a uniform way.

4.1 Notation

The following notation is used:

- $\mathcal{A}$ is an infinite set of all atomic formulas;
- $\mathcal{P}$ is a fixed finite subset of $\mathcal{A}$;
- $\mathcal{L}(\mathcal{X})$ is a set of all formulas built using atomic formulas from $\mathcal{X}$, where $\mathcal{X} \subseteq \mathcal{A}$. Furthermore:
  - the formulas are considered to be in the disjunctive normal form:
  $$\varphi = \bigvee_{i \in I} \bigwedge_{j \in J_i} \varphi_{ij}, \text{ where every } \varphi_{ij} \text{ is atomic}$$
  - $\mathcal{L}(\mathcal{X}, \sqsubseteq)$ is a free distributive lattice of $\mathcal{X}$ ordered by $\sqsubseteq$.
- $[c]_R$ denotes the equivalency class containing some element c, i.e., set of those elements which are equivalent (wrt. the equivalence relation R) to c;

- C/R denotes the factor set consisting of the equivalence classes of R;
- $Dom(f)$ is the domain of the function f;
- $Ran(f)$ is the range of the function f;

4.2 Framework

In our framework, the abstraction functions and order relations are referred by the index k. In this work, we introduce three different abstraction functions together with the corresponding concrete order relations. Therefore, the index k is supposed to be from $\{1, 2, 3\}$.

In order to be able to use the abstraction functions in Method I, we need to define the following:

- concrete order relation $\leq_k$ over formulas from $\mathcal{L}(\mathcal{A})$;
- equivalence relation $\sim_k$ on $\mathcal{L}(\mathcal{A})$;
- abstraction function $\alpha_k^{\mathcal{P}}$ parameterized by $\mathcal{P}$; [1]
- concretization function $\gamma_k^{\mathcal{P}}$ also parameterized by $\mathcal{P}$.

We show that each pair of functions $\langle \alpha_k^{\mathcal{P}}, \gamma_k^{\mathcal{P}} \rangle$ is a Galois connection. We also generalize the relative completeness result from [2] and show that Method I is complete wrt. Method II if we instantiate the order relation by $\leq_k$ and pair of abstraction and concretization functions by $\langle \alpha_k^{\mathcal{P}}, \gamma_k^{\mathcal{P}} \rangle$.

4.2.1 Order Relations

We define the order relations $\leq_k$ on formulas from $\mathcal{L}(\mathcal{A})$ as implication provability by a theorem prover. We have that $\varphi \leq_k \varphi'$ if the theorem prover corresponding to the ordering $\leq_k$ can show the validity of implication $\varphi \vdash \varphi'$.

The theorem provers for $\leq_k$ are simulated by a given theorem prover according to the definitions for $\leq_k$ that are given below. The given theorem prover must be able to show the validity of the following implication for atomic formulas φ and φ':

$$\begin{aligned} &\varphi \wedge \varphi' \vdash \varphi \\ &\varphi \vdash \varphi \vee \varphi' \end{aligned} \tag{4.1}$$

[1] If the set of atomic predicates $\mathcal{P}$ is known from the context it might be omitted, i.e., we could write α_k instead of $\alpha_k^{\mathcal{P}}$.

We give a description how to simulate the theorem prover for the ordering $\leq_k$ over the formula structure: for conjunctions, and for disjunctions. Each conjunct is an atomic formula, and each disjunct is a conjunction. We introduce a new notation: less$_{relation}$ means "less wrt. *relation*". The same is assumed for "greater".

The order relations $\leq_k$ have a common definition for disjunctions. Note that we use local subsumption test [12], i.e., the disjuncts are treated independently from each other. Thus, we obtain a weaker but less expensive test which amounts to testing entailment between quadratically many combinations of conjunctions:

$$\bigvee_{i \in I} \varphi_i \leq_k \bigvee_{j \in J} \varphi'_j \quad \text{iff} \quad \forall i \in I\ \exists j \in J \,.\, \varphi_i \leq_k \varphi'_j$$

The definitions for conjunctions differ for different order relations.

Definition 1 (Order relation $\leq_1$). *For conjunctions of atomic formulas the relation $\leq_1$ is defined as equality between the conjuncts:*

$$\bigwedge_{i \in I} \varphi_i \leq_1 \bigwedge_{j \in J} \varphi'_j \quad \textit{iff} \quad \forall j \in J\ \exists i \in I \,.\, \varphi_i \equiv \varphi'_j$$

Definition 2 (Order relation $\leq_2$). *For conjunctions of atomic formulas the relation $\leq_2$ is defined as implication provability between the conjuncts (conjuncts are treated independently from each other):*

$$\bigwedge_{i \in I} \varphi_i \leq_2 \bigwedge_{j \in J} \varphi'_j \quad \textit{iff} \quad \forall j \in J\ \exists i \in I \,.\, \varphi_i \vdash \varphi'_j$$

Definition 3 (Order relation $\leq_3$). *For conjunctions of atomic formulas the relation $\leq_3$ is defined as the implication provability between the conjunction on the left-hand side and the conjuncts on the right-hand side:*

$$\bigwedge_{i \in I} \varphi_i \leq_3 \bigwedge_{j \in J} \varphi'_j \quad \textit{iff} \quad \forall j \in J \,.\, \bigwedge_{i \in I} \varphi_i \vdash \varphi'_j$$

Here are some examples of order relations $\leq_k$. Note that it is important to distinguish these relations from the order relation $\leq$ over rational and real numbers.

Example 1 (Order relation $\leq_1$).

$$\begin{array}{rcl}
(X \leq 1) & \leq_1 & (X \leq 1) \\
(X \leq 1 \wedge Y \geq 10) & \leq_1 & (X \leq 1) \\
(X \leq 1) & \not\leq_1 & (X \leq 2) \\
(X \leq 1) & \not\leq_1 & (X \leq 1 \wedge Y \geq 10)
\end{array}$$

Example 2 (Order relation $\leq_2$).

$$\begin{array}{rcl} (X \leq 1) & \leq_2 & (X \leq 10) \\ (X \leq 1 \wedge Y \geq 10) & \leq_2 & (X \leq 10) \\ (X \leq 1 \wedge X \geq 1) & \not\leq_2 & (X = 1) \end{array}$$

Example 3 (Order relation $\leq_3$).

$$(X \leq 1 \wedge X \geq 1) \quad \leq_3 \quad (X = 1)$$

Note that the examples of $\leq_1$ are also examples of $\leq_2$ and $\leq_3$; similarly, the examples of $\leq_2$ are examples of $\leq_3$. The order relations $\leq_k$ can be compared with each other.

Lemma 1 (Relation between $\leq_k$). *Let φ, and φ' be formulas from $\mathcal{L}(\mathcal{A})$. There are the following relations between the orderings $\leq_k$:*

$$\varphi \leq_1 \varphi' \quad \Rightarrow \quad \varphi \leq_2 \varphi' \quad \Rightarrow \quad \varphi \leq_3 \varphi'$$

Proof. Because the order relation $\leq_k$ share the same definition for disjunctions, we only show the above implications between conjunctions of atomic formulas, i.e., for $\bigwedge_{i \in I} \varphi_i$ and $\bigwedge_{j \in J} \varphi'_j$.

We have the following implications, which follow from the requirements to the theorem prover (4.1):

$$\begin{array}{rcl} \varphi_i \equiv \varphi'_j & \Rightarrow & \varphi_i \vdash \varphi'_j \\ \varphi_i \vdash \varphi'_j & \Rightarrow & \bigwedge \ldots \varphi_i \ldots \vdash \varphi'_j \end{array}$$

Thus, we have the following implications:

$$\begin{array}{rc} \forall j \in J \; \exists i \in I \;.\; \varphi_i \equiv \varphi'_j & \Rightarrow \\ \forall j \in J \; \exists i \in I \;.\; \varphi_i \vdash \varphi'_j & \Rightarrow \\ \forall j \in J \;.\; \bigwedge_{i \in I} \varphi_i \vdash \varphi'_j & \end{array}$$

Therefore, we conclude that $\varphi \leq_1 \varphi' \Rightarrow \varphi \leq_2 \varphi' \Rightarrow \varphi \leq_3 \varphi'$. $\square$

Note that the implications in other direction do not hold in general case.

The abstract order relation of the free distributive lattice $\mathcal{L}(\mathcal{P}, \sqsubseteq)$ is defined as follows.

Definition 4 (Order relation $\sqsubseteq$). *For conjunctions of atomic formulas, the relation $\sqsubseteq$ is defined as:*

$$\bigwedge_{i \in I} \varphi_i \sqsubseteq \bigwedge_{j \in J} \varphi'_j \quad \textit{iff} \quad \forall j \in J \; \exists i \in I \;.\; \varphi_i \equiv \varphi'_j$$

The order relations $\leq_1$ and $\sqsubseteq$ coincide. Furthermore, the abstract order relation $\sqsubseteq$ implies the concrete order relation $\leq_k$:

$$\varphi \sqsubseteq \varphi' \quad \Rightarrow \quad \varphi \leq_k \varphi' \tag{4.2}$$

This is a consequence from Lemma 1 and the fact that $\sqsubseteq$ and $\leq_1$ coincide.

4.2.2 Equivalence Relations

The equivalence relations $\sim_k$ over the set of formulas $\mathcal{F}(\mathcal{A})$ define the equivalence classes $[\,]_{\sim_k}$ and factor sets $/\sim_k$.

Definition 5 (Equivalence relation $\sim_k$). *Let φ and φ' be two formulas from the set $\mathcal{F}(\mathcal{A})$. The equivalence relation $\sim_k$ is defined as:*

$$\varphi \sim_k \varphi' \quad \textit{iff} \quad \varphi \leq_k \varphi' \textit{ and } \varphi \geq_k \varphi'$$

The abstraction and concretization functions are defined on equivalence classes of formulas (as defined above).

4.2.3 Abstraction and Concretization Functions

An abstraction function and a concretization function are constructed for each concrete order relation $\leq_k$. These functions are parameterized by a finite set of atomic formulas $\mathcal{P}$ and are denoted as $\alpha_k^{\mathcal{P}}$ and $\gamma_k^{\mathcal{P}}$. The index k ranges over $1, 2, 3$.

Abstraction Functions

Two types of notation for abstraction functions are used: "μ" and "constructive" notation. The "μ" notation shows the meaning of the abstraction function. The "constructive" notation reflects the way how the abstraction function is supposed to be computed.

In "μ"-notation all abstraction function share the following definition.

Definition 6 (Abstraction function $\alpha_k^{\mathcal{P}}$ in "μ" notation). *$\alpha_k^{\mathcal{P}}$ is defined as a function of $\mathcal{F}(\mathcal{A})/\sim_k$ into $\mathcal{L}(\mathcal{P})$. Let φ be a formula from $\mathcal{F}(\mathcal{A})$. The abstraction $\alpha_k^{\mathcal{P}}([\varphi]_{\sim_k})$ is the least$_{\sqsubseteq}$ formula in the lattice $\mathcal{L}(\mathcal{P})$ which is greater$_{\leq_k}$ then φ:*

$$\alpha_k^{\mathcal{P}}([\varphi]_{\sim_k}) = \mu\varphi' \in \mathcal{L}(\mathcal{P}, \sqsubseteq) \,.\, \varphi \leq_k \varphi'$$

The lattice $\mathcal{L}(\mathcal{P})$ is complete (because it is finite), hence, the abstraction always exists.

Each abstraction function $\alpha_k^{\mathcal{P}}$ has its own "constructive" characterization. Recall that we use the "constructive" characterization in the model checker to compute an abstraction function. In the "constructive" notation we omit the explicit notation of the equivalence classes to keep the notation simple. In the "constructive" notation the abstraction functions $\alpha_k^{\mathcal{P}}$ are described as follows:

$$\begin{aligned} \alpha_1^{\mathcal{P}}(\bigvee_{i\in I}\bigwedge_{j\in J_i}\varphi_{ij}) &= \bigvee_{i\in I}\bigwedge\{\varphi\in\mathcal{P} \mid \exists j\in J_i\,.\,\varphi_{ij}\equiv\varphi\} \\ \alpha_2^{\mathcal{P}}(\bigvee_{i\in I}\bigwedge_{j\in J_i}\varphi_{ij}) &= \bigvee_{i\in I}\bigwedge\{\varphi\in\mathcal{P} \mid \exists j\in J_i\,.\,\varphi_{ij}\vdash\varphi\} \\ \alpha_3^{\mathcal{P}}(\bigvee_{i\in I}\bigwedge_{j\in J_i}\varphi_{ij}) &= \bigvee_{i\in I}\bigwedge\{\varphi\in\mathcal{P} \mid \bigwedge_{j\in J_i}\varphi_{ij}\vdash\varphi\} \end{aligned} \tag{4.3}$$

Here are some examples of application of the abstraction functions with various parameter sets.

Example 4 (Abstraction functions). The following table contains results of applying the abstraction functions $\alpha_k^{\mathcal{P}}$ on different formulas φ for different parameter sets $\mathcal{P}$. Note, sometimes the result is the empty conjunction $\wedge\emptyset$ which is the top element in $\mathcal{L}(\mathcal{P})$ and means `true`.

φ	$\alpha_1^{\mathcal{P}}(\varphi)$	$\alpha_2^{\mathcal{P}}(\varphi)$	$\alpha_3^{\mathcal{P}}(\varphi)$
$\mathcal{P}=\{X\le 10, X\le 5, X=0\}$			
$X=5$	$\wedge\emptyset$	$X\le 10\wedge X\le 5$	$X\le 10\wedge X\le 5$
$X=0$	$X=0$	$X\le 10\wedge X\le 5\wedge X=0$	$X\le 10\wedge X\le 5\wedge X=0$
$\mathcal{P}=\{X=0\}$			
$X\le 0\wedge X\ge 0$	$\wedge\emptyset$	$\wedge\emptyset$	$X=0$

Table 4.1: Abstraction functions examples.

We show that the "constructive" and the "μ" characterizations are equivalent. This gives us an opportunity to reason about the abstraction functions in a uniform way, i.e., every proof that holds for the "μ" characterization is also a proof for each "constructive" characterization. It is not difficult to see that all "constructive" characterizations have a common structure:

$$\alpha_k^{\mathcal{P}}(\bigvee_{i\in I}\bigwedge_{j\in J_i}\varphi_{ij}) = \bigvee_{i\in I}\bigwedge\{\varphi' \in \mathcal{P} | \bigwedge_{j\in J_i}\varphi_{ij} \leq_k \varphi'\}$$

We use this fact to prove the following lemma.

Lemma 2 (Equivalence between the "constructive" and the "μ" characterizations of $\alpha_k^{\mathcal{P}}$). *The "constructive" characterizations of the abstraction functions $\alpha_k^{\mathcal{P}}$ are equivalent to the "μ" characterization.*

Proof.

$$\begin{aligned}\alpha_k^{\mathcal{P}}(\bigvee_{i\in I}\bigwedge_{j\in J_i}\varphi_{ij}) &= \bigvee_{i\in I}\bigwedge\{\varphi' \in \mathcal{P} | \bigwedge_{j\in J_i}\varphi_{ij} \leq_k \varphi'\} \\ &= \bigvee_{i\in I}(\mu\varphi' \in \mathcal{L}(\mathcal{P},\sqsubseteq) \ . \bigwedge_{j\in J_i}\varphi_{ij} \leq_k \varphi') \\ &= \mu\varphi' \in \mathcal{L}(\mathcal{P},\sqsubseteq) \ . \bigvee_{i\in I}\bigwedge_{j\in J_i}\varphi_{ij} \leq_k \varphi'\end{aligned}$$

□

Concretization Functions

All concretization functions $\gamma_k^{\mathcal{P}}$ share the following definition.

Definition 7 (Concretization function $\gamma_k^{\mathcal{P}}$). *$\gamma_k^{\mathcal{P}}$ is defined as a function of $\mathcal{L}(\mathcal{P})$ into $\mathcal{F}(\mathcal{A})/_{\sim_k}$. Let φ be a formula from $\mathcal{L}(\mathcal{P})$. The concretization $\gamma_k^{\mathcal{P}}(\varphi)$ is a equivalence class of the greatest$_{\leq_k}$ formula in $\mathcal{F}(\mathcal{A})$ which is less$_{\sqsubseteq}$ than φ:*

$$\gamma_k^{\mathcal{P}}(\varphi) = [\varphi']_{\sim_k} \ . \ \nu\varphi' \in \mathcal{F}(\mathcal{A}) \ . \ \varphi' \sqsubseteq \varphi$$

Later on, we keep it implicit that the range of the concretization function $\gamma_k^{\mathcal{P}}$ is a quotient lattice.

Corollary 1 (The concretization function $\gamma_k^{\mathcal{P}}$ means identity). *The concretization function $\gamma_k^{\mathcal{P}}$ means identity:*

$$\gamma_k^{\mathcal{P}}(\varphi) = \varphi$$

Proof. The domain of $\gamma_k^{\mathcal{P}}$ is a set of atomic formulas $\mathcal{P}$ that is a subset of $\mathcal{A}$. Hence, every formula from $\mathcal{L}(\mathcal{P})$ is an element of $\mathcal{L}(\mathcal{A})$. By (4.2) if

$\varphi' \sqsubseteq \varphi$ then $\varphi' \leq_k \varphi$. Moreover, φ itself is the greatest$_\sqsubseteq$ formula that is less$_\sqsubseteq$ then φ:

$$\begin{array}{rcl} & & \varphi \sqsubseteq \varphi \\ \forall \varphi' \in \mathcal{L}(\mathcal{A}) : \varphi' \sqsubseteq \varphi & \Rightarrow & \varphi' \leq_k \varphi \end{array}$$

Hence, $\gamma_k^{\mathcal{P}}(\varphi)$ is φ. □

4.2.4 Galois Connection

The setup of a Galois connection between concrete and abstract domains guarantees the safety of the approximation, and therefore, the correctness of the model checking method. A Galois connection $\langle \alpha, \gamma \rangle$ ensures that the loss of information in the abstraction process is sound, and that the concretization process introduces no loss of information [8]:

$$\begin{array}{l} \forall x \in L \,.\, x \leq \gamma \circ \alpha(x) \\ \forall \overline{x} \in \overline{L} \,.\, \alpha \circ \gamma(\overline{x}) \sqsubseteq \overline{x} \end{array}$$

We prove that the abstraction function $\alpha_k^{\mathcal{P}}$ together with the concretization function $\gamma_k^{\mathcal{P}}$ defines a Galois connection. Note that the following theorems hold for each pair of abstraction and concretization function defined as (6) and (7) and do not depend on the "constructive" characterization.

In order to show the Galois connection $\langle \alpha_k^{\mathcal{P}}, \gamma_k^{\mathcal{P}} \rangle$, the following equivalence must be proved:

$$\forall \varphi \in Dom(\alpha_k^{\mathcal{P}})\ \forall \varphi^\# \in Ran(\alpha_k^{\mathcal{P}}) \,.\, \alpha_k^{\mathcal{P}}(\varphi) \sqsubseteq \varphi^\# \ \Leftrightarrow\ \varphi \leq_k \gamma_k^{\mathcal{P}}(\varphi^\#) \tag{4.4}$$

Theorem 1 ($\langle \alpha_k^{\mathcal{P}}, \gamma_k^{\mathcal{P}} \rangle$ builds a Galois connection). *The pair $\langle \alpha_k^{\mathcal{P}}, \gamma_k^{\mathcal{P}} \rangle$ is a Galois connection.*

Proof. In order to prove the theorem we need to show that equivalence 4.4 holds for both directions. Let φ and $\varphi^\#$ be elements of $\mathcal{L}(\mathcal{A})$ and resp. $\mathcal{L}(\mathcal{P})$.

Considering the implication to the right:

$$\alpha_k^{\mathcal{P}}(\varphi) \sqsubseteq \varphi^\# \quad \Rightarrow \quad \varphi \leq_k \gamma_k^{\mathcal{P}}(\varphi^\#)$$

By the construction of $\alpha_k^{\mathcal{P}}$, we have $\varphi \leq_k \alpha_k^{\mathcal{P}}(\varphi)$. Applying (4.2):

$$\alpha_k^{\mathcal{P}}(\varphi) \sqsubseteq \varphi^\# \quad \Rightarrow \quad \alpha_k^{\mathcal{P}}(\varphi) \leq_k \varphi^\#$$

By the transitivity of $\leq_k$ we have that $\varphi \leq_k \varphi^{\#}$. The concretization function means identity, hence, $\varphi \leq_k \gamma_k^{\mathcal{P}}(\varphi^{\#})$.

Let's prove the implication to the left:

$$\alpha_k^{\mathcal{P}}(\varphi) \sqsubseteq \varphi^{\#} \quad \Leftarrow \quad \varphi \leq_k \gamma_k^{\mathcal{P}}(\varphi^{\#})$$

We have $\varphi \leq_k \varphi^{\#}$ and $\varphi \leq_k \alpha_k^{\mathcal{P}}(\varphi)$. By the definition, $\alpha_k^{\mathcal{P}}(\varphi)$ is the least$_{\sqsubseteq}$ element in $\mathcal{L}(\mathcal{P})$ that is greater$_{\leq_k}$ then φ. Hence, $\alpha_k^{\mathcal{P}}(\varphi) \sqsubseteq \varphi^{\#}$. $\square$

4.2.5 Relative Completeness

There is a relative completeness property described in [2] which is expressed by the following theorem.

Theorem 2 (Relative Completeness of Abstract Backward Iteration with Backward Refinement). *For every program,* backward *abstract fixpoint iteration with* backward *abstraction refinement is guaranteed to terminate with success if there exists an oracle such that backward iteration with oracle-guided widening, terminates with success.*

The preconditions of this theorem are the following:

- the order relation on formulas $\leq$ is defined as implication provability by a theorem prover[2];
- the abstraction function is defined as

$$\alpha^{\mathcal{P}}(\varphi) = \mu\varphi' \in \mathcal{L}(\mathcal{P}, \sqsubseteq) \, . \, \varphi \leq \varphi'$$

- the concretization function $\gamma^{\mathcal{P}}$ means identity.

We observe that the relative completeness property holds if we generalize the preconditions. Namely, it holds if we instantiate the order relation $\leq$ by the order relation $\leq_k$, and take $\alpha_k^{\mathcal{P}}$ and $\gamma_k^{\mathcal{P}}$ as abstraction and concretization functions.

In order to show the validity of the generalization, we show that the preconditions of Theorem 2 hold in the generalized instantiation. We have that if the ordering $\leq$ is instantiated by $\leq_k$ then the abstraction function $\alpha_k^{\mathcal{P}}$ has the same meaning as $\alpha^{\mathcal{P}}$ by Lemma 2, and the meaning of $\gamma_k^{\mathcal{P}}$ is identity. Therefore, we can conclude that Theorem 2 holds in the generalized setting.

[2]It is required that the theorem prover must be able to prove the implications (4.1).

4.2.6 Complexity Estimations

We make the following complexity estimation using the arithmetic model of computation [19]: the number of steps needed to compute the abstraction $\alpha_k^{\mathcal{P}}(\varphi)$, and how fast the set of atomic predicates $\mathcal{P}$ grows during the abstraction refinement process.

We estimate the number of steps needed to compute the abstraction $\alpha_k^{\mathcal{P}}(\varphi)$, defined by the "constructive" characterization (4.3). We take the size of the operand formula φ and the number of elements in $\mathcal{P}$ as parameters:

- $|\mathcal{P}|$ denotes the number of elements in $\mathcal{P}$,
- $|\varphi|$ is the number of atomic formulas in a formula $\varphi = \bigvee_{i\in I} \bigwedge_{j\in J_i} \varphi_{ij}$, where the size of the biggest conjunction is denoted by $|J|$.

$$
\begin{aligned}
|\varphi| &= \sum_{i\in I} |J_i| \\
|J| &= \max_{i\in I} |J_i|
\end{aligned}
$$

We assume that the entailment check $\varphi_1 \wedge \cdots \wedge \varphi_m \leq \varphi$ can be done in $f(m)$ steps, and that the membership check whether an atomic formula is in $\mathcal{P}$ can be done in $O(1)$ using a hash-table data structure [21]. We can use the string representation of an atomic formula as the input for the hash-function.

The complexity estimations are listed in Table 4.2.

	Construction Steps	**Complexity**
$\alpha_1^{\mathcal{P}}$	For each φ_{ij} we check whether it is in $\mathcal{P}$.	$\Theta(\lvert\varphi\rvert)$
$\alpha_2^{\mathcal{P}}$	For each φ_{ij} we take all φ' from $\mathcal{P}$ such that $\varphi_{ij} \vdash \varphi'$ holds.	$\Theta(\lvert\varphi\rvert \cdot \lvert\mathcal{P}\rvert)$
$\alpha_3^{\mathcal{P}}$	For each $\bigwedge_{j\in J_i} \varphi_{ij}$ we take all φ' from $\mathcal{P}$ such that $\bigwedge_{j\in J_i} \varphi_{ij} \vdash \varphi'$ holds.	$O(\lvert I\rvert \cdot \lvert\mathcal{P}\rvert \cdot f(\lvert J\rvert))$

Table 4.2: Complexity estimations for the abstraction functions $\alpha_k^{\mathcal{P}}$.

There is the following estimation for the complexity of the entailment check $f(m)$. Using the ellipsoid method outlined in [19] we can decide whether a system of linear inequalities $Ax \leq b$ with integer coefficients is solvable over the real or rational numbers. The method stops after

$$N = 2n((2n+1)\langle A\rangle + n\langle b\rangle - n^3)$$

iterations, where $\langle . \rangle$ denotes the encoding length. In praxis there are specialized constraint solvers that can do entailment check very fast using various optimization techniques.

We make an estimation of the number of atomic formulas in the set $\mathcal{P}$ after n refinement iteration using the following parameters:

- n is the number of the refinement iterations done;
- G is the size of the biggest guard in the program;
- N is the maximal number of commands that lead to a particular location.

We observe that during a transition $\varphi^{n+1} = \mathsf{pre}(\varphi^n)$ [3] every atomic formula in φ^n can be updated, and a guard formula is added. Furthermore, the transition is done for at most N commands. Hence, we obtain the following estimations:

$$\begin{aligned} O(|\varphi^{n+1}|) &= O((G + |\varphi^n|) \cdot N) \\ &= O(G \cdot N + \ldots + G \cdot N^n + |\mathsf{nonInit}| \cdot N^n) \\ &= O(N^n) \end{aligned}$$

Hence, the upper approximation for the size of $\mathcal{P}^n$ is $O(n \cdot N^n)$. The parameter set $\mathcal{P}$ grows exponentially fast for programs with branching control flow structure, but if every program label has only one transition associated with it (i.e., straight-line program with $N = 1$) then the size of $\mathcal{P}$ grows linear in the number of refinement iteration.

[3] $\varphi^0 = \mathsf{nonInit}$

Chapter 5

Abstraction Refinement Model Checker

We implemented Method I, described in Section 3.3, in a model checker called ARMC[1]. In this chapter we describe its design and implementation.

5.1 Design

One of the key features of ARMC is that it can be easily modified and extended. This is achieved using a modular architecture. The major components of ARMC are given in Figure 5.1. The components have the fol-

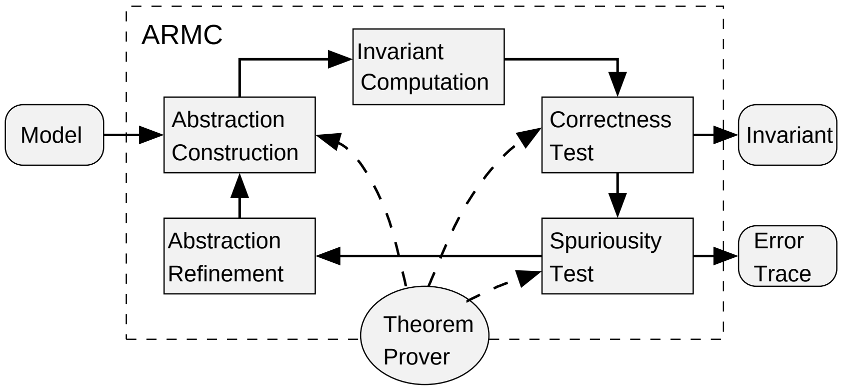

Figure 5.1: ARMC architecture.

[1] **A**bstraction **R**efinement **M**odel **C**hecker

lowing functionality:

Abstraction Construction The abstraction function $\alpha_k^{\mathcal{P}_n}$ is constructed by this component. The atomic formulas that parameterize the construction are provided by the component **Abstraction Refinement**. The component **Theorem Prover** is requested to prove the implication validity during the construction of the abstraction.

Invariant Computation This component computes an invariant, i.e, it finds the least fixpoint of the abstract transition operator constructed by the component **Abstraction Construction**.

Correctness Test The correctness of the found invariant is tested using the component **Theorem Prover**. The model checker halts if the correctness of the invariant is proved.

Spuriousity Test This test finds out whether the found invariant proved to be incorrect is a witness or a spurious error trace. **Theorem Prover** component is requested to do satisfiability checks during the traverse of the error trace. If the invariant is a witness error trace then the model checker halts.

Abstraction Refinement If the error trace is spurious, the abstraction is refined by this component. It iterates the transition operator and collects the resulting predicates.

Theorem Prover The theorem prover is an external component that is requested by other components to prove the implication validity, or to do a satisfiability check. Using different theorem provers with different precision we can adjust the precision of the model checker.

The verification result is an invariant that implies the correctness of the system, or a witness error trace.

5.2 Implementation

The model checker is written in SICStus Prolog [15]. As the given theorem prover it uses a constraint solver CLP(Q, R) [14] which is a part of the SICStus Prolog distribution.[2]

Prolog programming language is a good choice for writing programs dealing with symbolic computation and rapid prototyping [20, 17]. Such

[2] `http://www.sics.se/sicstus`

features are important for an implementation of the experimental symbolic model checker software. SICStus Prolog is a system with powerful libraries, and a rich set of programming tools including a debugger, profiler, etc.

5.2.1 Model Representation

The model of the system that we want to verify and a correctness specification are represented as Prolog terms. The term structure is adopted from DMC, thus, it is possible to reuse models written for DMC as well as tools that create output in DMC format.

Now we describe the model encoding.

Unsafe States A state description, i.e., the state variables and the location label, is encoded as a state term of the following structure:

$$\mathtt{p}(PC, X_1, \ldots, X_n)$$

where PC is the location label, and $X_1, \ldots, X_n$ are the system variables in this location. A state term together with a set of constraints over its variables encodes a set of program states in some program location.

The set of backward reached states (as well as the set unsafe) is encoded as term of the following structure:

$$\mathtt{s}(Level, State, Constraint, ID, (RuleID, FactID))$$

The description of the term components can be found in Table 5.1.

Transition Rules The encoding of a transition rule is the following:

$$\mathtt{r}(OldState, NewState, Action, ID)$$

The term components have the meaning shown in Table 5.2.

As described in Section 3.1, the syntactic iteration of the transition function requires representation of a command as a combination of a guard and updates. Therefore, there is another term representation of the transition rules which is used in the syntactic iterations:

$$\mathtt{r}(OldState, NewState, Action, ID, Guards, Updates)$$

The partition of *Action* into *Guards* and *Updates* is done automatically according to the following rules:

Component	Description
Level	Number of the iteration step at which this set of states was added to the set of unsafe states.
State	State term containing the location label and the state variables.
Constraint	Formula over the state variables.
ID	Unique index; the unsafe states are numbered in order to be able to construct error traces.
RuleID	Index of the transition rule which leads to this unsafe state.
FactID	Index of the unsafe state which is the successor of this state.

Table 5.1: Components of term representing unsafe.

Component	Description
OldState	Start state of the transition (with unprimed variables).
NewState	End state of the transition (with primed variables).
Action	Formula over the unprimed and primed variables.
ID	Unique index; all rules are numbered, so we can easily construct (error) traces.

Table 5.2: Components of term representing transition rules.

- A constraint over unprimed variables belongs to the guard;
- A constraint of the form `X' = ...` where `X'` is a primed variable is an update.

nonInit The set of states representing nonInit is encoded as a set of the following terms:

$$\mathsf{b}(State, Constraint, ID)$$

where the meaning of the term components is given in Table 5.3:

Component	Description
State	State description term.
Constraint	Constraint over the state variables.
ID	Unique index.

Table 5.3: Components of term representing nonInit.

Example Consider the transition system from Figure 3.1. Let `error` be the initial unsafe state. We encode the system transitions as the following terms:

$$
\begin{array}{lcr}
\texttt{r}(\texttt{p}(\texttt{init}, X), \texttt{p}(\texttt{plus}, X'), & \{X' = 0\}, & 1). \\
\texttt{r}(\texttt{p}(\texttt{plus}, X), \texttt{p}(\texttt{plus}, X'), & \{X < 3, X' = X + 1\}, & 2). \\
\texttt{r}(\texttt{p}(\texttt{plus}, X), \texttt{p}(\texttt{error}, X), & \{X >= 3\}, & 3).
\end{array}
$$

nonInit consists of all states that are not in the location labeled `init`, thus, we have:

$$
\begin{array}{rl}
\texttt{b}(\texttt{plus}, & \{\}, 1). \\
\texttt{b}(\texttt{error}, & \{\}, 2).
\end{array}
$$

and the set of the initial unsafe states unsafe is:

$$\texttt{s}(0, \texttt{p}(\texttt{error}, X), \{\}, 1, (0, 0)).$$

5.2.2 Programming Issues

The main difficulty by programming ARMC was the implementation of the syntactic operations on terms that represent sets of states, guards, and updates. Such operations are unavoidable because of the "syntactic" definition of the transition operator. The main issue was to come up with non-binding predicates, e.g., a list membership test or an unifyability check which compare the terms syntactically. Our implementation of these predicates extensively uses the meta-logic predicates of Prolog.

To give an example how the mentioned difficulty was handled, let us write a predicate `expose(Var, Constraint, Expr)` which makes the information about the given variable `Var` explicit after the simplification of the constraint `Constraint` over this variable. The explicit representation is put into the variable `Expr`. The main difficulty consists in making the value of the variable explicit if it becomes binded.[3]

[3] The similar functionality has ARMC predicate `simplify_ex/2` in module **Utilities**.

A naive implementation would look like in Figure 5.2. It works for con-

```
expose_naive(Var, C, R) :-
              simplify(C, S),
              {R} = S.
```

Figure 5.2: Predicate `expose_naive/3`.

straints which simplification does not cause the binding of the constrained variable, e.g., for C ={`X > 1`}, but it fails to give a representation of the binded variable, e.g.,for C ={`X = 1`}, because `simplify({X = 1}, R)` produces an empty term `R = {}` and binds the variable `X` to 1. To be able to handle such cases, we suggest another version of `expose/3`. The clauses are shown in Figure 5.3.

```
A.1  expose(Var, C, R) :-
  2          simplify(C, S),
  3          S \== {},
  4          !,
  5          {R} = S.

B.1  expose(Var, C, _) :-
  2          simplify(C, R),
  3          R == {},
  4          bb_put(value, Var),
  5          fail.

C.1  expose(Var, _, R) :-
  2          bb_get(value, Val),
  3          functor(R, '=:=', 2),
  4          arg(1, R, Var),
  5          arg(2, R, Val).
```

Figure 5.3: Predicate `expose/3`.

The predicate `expose/3` has three clauses A, B, and C, which are responsible for capturing different result possibilities of the constraint simplification. Now, we go through the clauses and explain how they work.

If the constraint simplification (line A.2) does not lead to variable binding, i.e., if the constraint does not become empty (A.3), then the result is a term built from the simplified constraint (A.5). We have to put the cut symbol (A.4) to prevent the backtracking.

If the simplification binds the variable to some value then the resulting term is empty (B.3). The computed value is put on a blackboard[4] (B.4), and the clause C is called (B.5).

The variable `Var` is not binded in the context of the clause C, but its value has been already computed in clause B, and put onto the blackboard. Thus, the rest is straightforward: get the value from the blackboard (C.2), and construct its representation (C.3, C.4, and C.5).

The test runs for different cases are shown in Table 5.4.

`| ?- expose(X, {X > 0}, R).`	`| ?- expose(X, {X = 0}, R).`
`R = X>0.0 ?`	`R = X=:=0.0 ?`
`yes`	`yes`

Table 5.4: Applications of `expose/3`.

5.3 Short User's Manual

In this section we briefly describe how to load ARMC code into the SICStus Prolog interpreter and start working with it.

5.3.1 Loading the ARMC Code

Go to the directory with the ARMC distribution and start the SICStus Prolog interpreter.[5] Type "`[method1].`" followed by *Return* key in the interpreter prompt to read in the ARMC code. After the program modules are loaded, the model checker is ready to use.

5.3.2 Input File Layout

For technical reasons, a text file containing a system representation must have the following preamble:

[4]The blackboard is an intermediate predicate storage facility provided by SICStus Prolog.

[5]For more information about SICStus Prolog see [15]

```
:- multifile s/5.
:- multifile r/4.
:- multifile b/3.
```

The terms that describe the system can appear in the input file at any position after the preamble.

5.3.3 User Interface

The user interface of ARMC consists of a set of predicates which are listed in Table 5.5. The initialization parameters and their descriptions are given

Predicate	**Parameter(s)**	**Action**
`init`($File, Type, Opts$)	Name of the input file, the abstraction type, and the list of options.	Reads the input from the specified file, and initializes the model checker.
`check`		Starts the model checking process.
`print_all`		If called after the termination of `check`, prints information about the constructed atomic predicates, the abstract fixpoint, and the syntactic trace.
`print_error_trace`		Prints the concrete error trace.
`utils:path`(ID)	The id of a concrete unsafe state.	Prints a path in the concrete error trace starting from the given state and leading to unsafe.
`utils:path_a`(ID)	The id of an abstract unsafe state.	Prints a path in the abstract fixpoint starting from the given state and leading to unsafe.
`utils:stat`		Collects and prints the statistics for atomic predicates: the number of predicates used to express the fixpoint, and the number of generated predicates; partitioned by location labels.

Table 5.5: User interface predicates.

in Table 5.6.

Option	Value	Description
`atomics`	`global` or `local`	Determines if the set of atomic predicates should be partitioned by the location labels.
`cfp_chk`	`true` or `false`	If this option is set to `true` then the refinement iteration halts if the concrete fixpoint is reached during the spuriousness test and the safety of the concrete fixpoint will be checked. The default value is `(cfp_chk, true)`.

Table 5.6: Initialization parameters.

5.3.4 Profiling

It is possible to get information how often a particular ARMC predicate was called during the system verification. There is a graphical tool called Gauge, which is a part of SICStus Prolog environment, for viewing the profile data for the predicates covered by a given specification. This tool can be accessed using the following sequence of commands typed in the interpreter prompt:

- `prolog_flag(compiling,_,profiledcode).`
- `compile([utils, abstraction, fl, fs, fe, method1]).`
- `use_module(library(gauge)).`
- `view(_:_).`

The profile data is generated by running the model checker. The tool Gauge is described in [15].

Chapter 6

Experiments

We used ARMC to verify the safety of various computer programs that belong to the following classes of integer systems: imperative programs, communication protocols, and parameterized systems. Most of the examples are taken from the case studies described in [12] and [10].

We give a short description of the examples, their verification using different abstraction functions, and evaluate the effectiveness of the abstraction functions in verifying the programs. Verification statistics for each example can be found in Table 6.1. We also verified the examples using DMC and compared the execution times[1] (see Table 6.2). DMC can be considered as an implementation of Method II equipped with a deductive widening operator (as a simulation of the oracle-based operator).

6.1 Imperative Programs

We considered the following programs written in C programming language: **bpr.c** - the example program from [2]; **inssort.c** - insertion sort implementation taken from [18]; **selsort.c** - selection sort implementation, the model is taken from [12].

bpr.c The source code of this example is shown in Figure 6.1. The ARMC model is given in Appendix A.1. We check the unreachability of the program point with the label `error`.

Note that there is an infinite loop before the label `error`. This makes a verification using a plain fixpoint iteration impossible: a concrete exe-

[1] We used AMD Duron 1 GHz, Linux 2.4.0, and Sicstus Prolog 3.8.7.

```
foo() {
    int x, y, z;

L1: x = 0;
L2: while (x >= 0) {
        x = x + 1;
    }
L3: if (y == 25) {
L4:     if (y != 25) {
L5:         z = -1;
L6:         while (z != 0) {
                z = z - 1;
            }
error:      ;
        }
    }
}
```

Figure 6.1: Example program **bpr.c**.

cution of the program produces the following infinite sequence:

$$
\begin{array}{ll}
p(err, x, y, z) & \\
p(L6, x, y, z), & z = 0 \\
p(L6, x, y, z), & z = 1 \\
\dots &
\end{array}
$$

Verification using each abstraction function succeeded after 3 refinement steps.

inssort.c This program is an implementation of the insertion sorting algorithm, and its source code (see Figure 6.2) is taken from the book "Numerical Recipes in C" [18]. The model is given in Appendix A.2.

This program contains array operations within two loops: `for`, and `while` loop. We checked that the array bounds are not violated. Because the array operations are executed inside the loops, verification using a plain fixpoint iteration is not possible.

Verification using the abstraction function α_1 leaded to a non-terminating iteration. The abstraction functions α_2 and α_3 proved the safety after one refinement iteration.

```
void InsertionSort(int A[], int n) {
    int i, k, x;

    for(k = 1; k < n; k++) {
        x = A[k];
        i = k - 1;
        while (i >= 0 && A[i] > x) {
            A[i + 1] = A[i];
            i--;
        }
        A[i + 1] = x;
    }
}
```

Figure 6.2: Example program **inssort.c**.

6.2 Timed Automaton

There is a model of a coffee machine in form of a timed automaton[2] shown in Figure 6.3. The result of the translation of the automaton into a set of transition rules can be found in Appendix A.3.

We verified the model wrt. the following property: a cup of coffee is ready in 15 seconds after the customer pressed the button under condition that the cup of coffee has been paid.

We proved the property by showing that the the state labeled `init` is reachable under the following conditions:

- the coffee is ready in 15 seconds, i.e., $wait \leq 15$;
- the coffee was paid, i.e., $paid \geq price$.

Each abstraction produced the following witness trace after 3 refinement iteration:

$$\begin{array}{rl} p(coffee, c1, c2, e, wait, paid, price), & wait \leq 15 \\ \vdots & \\ p(init, c1, c2, e, wait, paid, price), & paid \geq price \end{array}$$

which proves the correctness of the model.

[2]Kindly provided by Jochen Hoenicke from the University of Oldenburg.

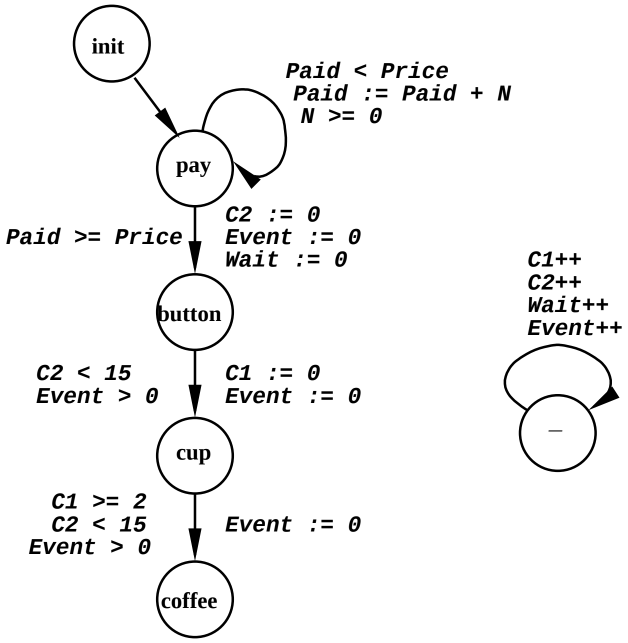

Figure 6.3: Coffee machine automaton and counter updates for each state.

6.3 Communication Protocols

We took the following examples considered in the case study in [12]: mutual exclusion and starvation freedom protocols: bakery and ticket; producer-consumer protocols: via bounded and unbounded buffer. We also verified Fischer's mutual exclusion protocol. We verified the protocols for systems consisting of 2 processes.

The source code of the examples (except Fischer's protocol) can be found at the following address: www.mpi-sb.mpg.de/˜delzanno/clp.html. The source code of Fischer's protocol is listed in Appendix A.4.

Verification using the abstraction function α_1 succeed only for Fischer's protocol. For other protocols, the abstraction functions α_2 and α_3 proved the safety.

6.4 Parameterized Systems

We verified the following parameterized cache coherence protocols using the approach described in [10]: Berkeley, Dragon, Firefly, Futurebus+,

Program	Size C/V	Abstraction R/I P/G					
		α_1		α_2		α_3	
bpr.c	9/3	3/6	4/6	3/6	6/6	3/6	6/6
inssort.c	9/3	$\uparrow$		1/5	9/11	1/5	9/11
selsort.c	7/4	$\uparrow$		2/5	10/14	2/5	10/14
coffee	10/6	3/7	15/15	3/7	15/15	3/7	15/15
bakery	25/2	$\uparrow$		4/14	21/25	4/14	21/25
ticket	19/4	$\uparrow$		4/9	7/7	4/9	7/7
fischer	41/3	1/1	4/4	1/1	4/4	1/1	4/4
bbuffer	5/6	$\uparrow$		0/0	1/1	0/0	3/3
ubuffer	9/4	$\uparrow$		0/0	1/1	0/0	1/1
berkeley	10/4	$\uparrow$		0/0	3/3	0/0	3/3
dragon	15/5	$\uparrow$		1/4	31/44	1/1	21/44
firefly	15/4	$\uparrow$		1/2	17/24	1/1	15/24
futurebus+	12/9	$\uparrow$		4/8	210/707	4/8	552/707
illinois	15/4	$\uparrow$		2/2	31/66	2/2	50/66
mesi	9/4	$\uparrow$		2/2	12/66	2/2	33/79
moesi	10/5	$\uparrow$		1/1	13/23	1/1	15/23
synapse	6/3	$\uparrow$		0/0	5/5	0/0	5/5

Table 6.1: Abstraction refinement statistics: C = number of clauses, V = number of variables (excluding program counter), R = number of refinement steps, I = number of steps during the last fixpoint iteration, P = number of different predicates in the invariant, G = number of generated predicates, $\uparrow$ = non-termination.

Illinois, MESI, MOESI, and Synapse N+1. These protocols are used to maintain data consistency in multiprocessor systems equipped with local caches. The safety property (i.e., data-consistency) was proved for systems consisting of any number of processors. The examples can be found at the following address: www.disi.unige.it/person/DelzannoG/protocol.html.

As for the communication protocols, verification using α_1 did not terminate. Applying the abstraction functions α_2 and α_3 the verification succeeded after a small number of the refinement steps.

6.5 Observations

We evaluate the effectiveness of the abstraction functions and the predicate generation procedure.

We can point the following reasons for the success of the verification:

Program	Abstraction			DMC
	α_1	α_2	α_3	
bpr.c	0.03	0.03	0.03	$\uparrow$
inssort.c	$\uparrow$	0.14	0.12	0.28
selsort.c	$\uparrow$	0.16	0.16	0.12
coffee	0.49	0.25	0.24	$\uparrow$
bakery	$\uparrow$	0.58	0.56	0.11
ticket	$\uparrow$	0.43	0.41	0.36
fischer	0.03	0.02	0.02	0.01
bbuffer	$\uparrow$	0.02	0.01	0.11
ubuffer	$\uparrow$	0.03	0.02	0.14
berkeley	$\uparrow$	0.06	0.06	0.8
dragon	$\uparrow$	5.66	1.65	8.05
firefly	$\uparrow$	1.64	0.93	0.41
futurebus+	$\uparrow$	$\sim$ 1 hour	$\sim$ 1 hour	38.09
illinois	$\uparrow$	4.98	4.5	1.11
mesi	$\uparrow$	2.13	1.8	0.33
moesi	$\uparrow$	1.39	1.11	0.59
synapse	$\uparrow$	0.04	0.04	0.01

Table 6.2: Execution time (in sec.) using different abstraction functions and DMC. $\uparrow$ = non-termination.

- The "syntactic" operator pre does not get stuck in an infinite loop, thus it can collect the predicates, needed to construct an invariant, that appear behind the loop;

- The abstraction function α_1 leaded to a non-terminating iteration in most cases because it is not powerful enough to approximate an infinite sequence like $x \geq 0, x \geq 1, x \geq 2, \ldots$ (similar sequences appear in most of the examples) by $x \geq 0$. The abstraction functions α_2 and α_3 can do such approximation. This explains why verification using the abstraction function α_1 did not terminate in most cases, but using α_2 and α_3 it did.

We observe that the verification using the abstraction function α_3 runs faster than using α_2, although α_3 is theoretically more expensive. This can be explained as follows: the theorem prover checks the validity of implication $\varphi_1 \wedge \ldots \varphi_n \vdash \varphi$ faster then the model checker iterates a loop containing the implication check $\varphi_i \vdash \varphi$ for $i = 1, \ldots, n$.

Verification using the abstraction function α_2 was successful for all examples. Thus, the loss of precision in the ordering $\leq_2$ relative to the ordering $\leq_3$ due to the assumption that conjuncts are independent, is not crucial for our examples.

Chapter 7

Related Work

There are different verification methods based on predicate abstraction combined with automated refinement and model checkers implementing them. In this chapter we relate our work to some of these methods.

The SLAM toolkit [3, 1] for statical analysis of C programs implements a model checking method which is close to Method I [2]. An abstract boolean program is constructed from a C program under abstraction induced by a fixed set of predicates by introducing a boolean variable for each predicate [1]. The boolean program is checked by computing the set of reachable states. This set is a set of bit vectors and corresponds to the invariant formula in Method I. If a spurious error trace is found the boolean program is refined by providing new predicates that are extracted from that trace.

The quality measure for the model checking method implemented in the SLAM toolkit is the relative completeness result from [2].

The BLAST toolkit [13] implements the concept of "lazy abstraction". Lazy abstraction continuously constructs and refines an abstract model of a program on demand, as requested by the model checker. Such abstraction procedure exhibits different parts of the model with different degrees of precision. The abstraction is refined "from the pivot state on". This is done by finding the first spurious (going forward) transition in the spurious error trace, and refining at this point by predicates obtained from the proof of the transition spuriousness produced by a proof-generating theorem prover.

There is a termination criterion introduced for this algorithm. It requires that (1) the transition system must have a finite trace equivalence, and (2) there does not exist a strictly increasing chain of sets of states constructed using predicates from the chosen predicate language. The precondition (2) is usually not satisfied if an infinite set of predicates is used. However, for a system with a finite trace equivalence a finite set of predicates may be chosen. E.g., this is the case for timed automata.

The ARMC model checker differs from the toolkits described above by focusing on providing a flexible evaluation platform for different abstraction functions. Concerning termination conditions, the relative completeness from [2] is a more ambitious criterion than the criterion from [13]. If the conditions (1) and (2) are satisfied (in particular, if the predicate language is finite), then Method I terminates (at the latest after all predicates are collected).

Another algorithm to compute an abstract program by means of syntactic program transformations is described in [16]. It starts with the atomic predicates in the specification formula and the original program. New predicates are obtained by a syntactic weakest liberal precondition (wlp) computation. The algorithm is complete, in that, if the concrete program has a finite abstraction wrt. to simulation (bisimulation) equivalence, the method can compute a finite simulation-equivalent (bisimulation-equivalent) abstract program. The algorithm is parameterized by the upper bound on the number of allowed iterations K which is not explicitly stated. In general case, it is unknown whether all necessary atomic predicates are already collected or not during a wlp computation of the algorithm. There is a restricted class of programs (e.g., programs with assignments of the form $X := Y$) on which the algorithm terminates with a bisimulation-equivalent abstract program.

In the area of hardware verification abstraction is performed by selecting a set of variables (over finite domains) and making them invisible, i.e., they are treated as inputs. The model is refined by making variables visible, i.e., by adding their logic into the model. There are different approaches to selecting a small set of variables to make visible. E.g., use the variable dependency graph, combination of sampling with Integer Linear Programming (ILP) or machine learning [4]. Using the variable dependency graph the candidates in the next refinement step are chosen among the invisible variables adjacent to currently visible variables using information from the counterexample. The ILP problem and the decision tree are constructed using samples of the states in which the spurious abstract error trace and the concrete error trace traversal disagree. The solution of the ILP problem or labeling of the decision tree gives a separating set for the next refinement step, i.e., a set of variables which logic makes the spurious transition impossible in the refined model.

Another approach to variable abstraction is to partition variables into variable clusters and construct an abstraction function (a surjection) for each cluster as described in [5]. The initial partitioning is defined by the predicates in the program specification. The abstraction function given by a surjection induces an equivalence relation on the domain of the program variables. The abstraction function is refined by partitioning a single equivalence class so that a spurious transition is eliminated after refinement. The partitioning is done using information obtained from "the pivot state" by separating the states that are reachable from the initial state from the states that induce the spurious transition.

The idea of using advanced techniques to solve the refinement problem [5, 4] could be also applied to software model checking with predicate abstraction, i.e., the question where new predicates come from should be solved not by simply picking out predicates appearing in the counterexample or the proof of its spuriousness, but by considering only "good" predicates that guarantee the elimination of the spurious error trace.

Chapter 8

Conclusion

To summarize the work done in this thesis:

- We generalized the work described in [2] for a variety of abstraction functions, also new ones.

- We implemented a tool which provides a platform for experimenting with abstraction functions.

- We implemented a variety of abstraction functions and experimentally evaluated their effectiveness in verifying the correctness of the given examples.

Open Problems There are some directions of possible future work:

- Improve Method I so that it will be complete wrt. Method II based on a more powerful widening operator than the one that just drops conjuncts (this would result in a theoretically more powerful algorithm).

- Optimize predicate generation procedure to eliminate the spuriousness by adding minimal number of new predicates (this should improve the practical performance of the model checker).

- Find a forward abstraction refinement procedure which satisfies the relative completeness property (and compare the effectiveness of forward- and backward-iteration based methods).

Acknowledgements I would like to thank my advisor Prof. Andreas Podelski for the interesting thesis subject, for his guidance and support. Thanks to the other members and students of AG2 at MPII for providing an excellent working environment, useful insights, and time. I thank Georg Jung for proofreading and comments on a preliminary version of this thesis. Very special thanks to my parents, for their support and encouragement.

Bibliography

[1] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In Tiziana Margaria and Wang Yi, editors, *Proceedings of TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 268–283. Springer-Verlag, 2001.

[2] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In Joost-Pieter Kaoen and Perdita Stevens, editors, *Proceedings of TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158–172. Springer-Verlag, 2002.

[3] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3. ACM Press, 2002.

[4] Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of CAV: Computer Aided Verification*, volume 2404 of *LNCS*, pages 265–279. Springer-Verlag, 2002.

[5] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of CAV: Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[6] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[7] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[8] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see http://www.di.ens.fr/˜cousot.).

[9] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[10] Giorgio Delzanno. Verification of Consistency Protocols via Infinite-State Symbolic Model Checking. In Tommaso Bolognesi and Diego Latella, editors, *FORTE*, volume 183 of *IFIP Conference Proceedings*, pages 171–186. Kluwer, 2000.

[11] Giorgio Delzanno and Andreas Podelski. Model Checking in CLP. In Rance Cleaveland, editor, *Proceedings of TACAS'99, the Second International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 223–239. Springer-Verlag, 1999.

[12] Giorgio Delzanno and Andreas Podelski. Constraint-based Deductive Model Checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[13] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[14] C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3.* Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[15] The Intelligent Systems Laboratory. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, PO Box 1263 SE-164 29 Kista, Sweden, October 2001. Release 3.8.7.

[16] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In *Proceedings of CAV: Computer Aided Verification*, volume 1855 of *LNCS*, pages 435–449. Springer Verlag, 2000.

[17] Richard A. O'Keefe. *The Craft of Prolog.* MIT Press, Cambridge;London, 1990.

[18] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P . Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, 1992.

[19] Alan Robinson and Andrei Voronkov, editors. *Handbook of automated reasoning. - Vol. 1.* Elsevier, 2001.

[20] Leon Sterling and Ehud Shapiro. *The Art of Prolog. Advanced Programming Techniques.* MIT Press, Cambridge;London, 1994.

[21] Jan van Leeuwen, editor. *Algorithms and complexity*, volume A of *Handbook of theoretical computer science.* MIT Press, 1990.

Appendix A

ARMC Models

A.1 bpr.c

```
:- multifile s/5.
:- multifile r/4.
:- multifile b/3.

r(p(init,X,Y,Z), p(l1,X,Y,Z),   {},               10).
r(p(l1,X,Y,Z),   p(l2,X1,Y,Z),  {X1=:= 0},         1).  % C1
r(p(l2,X,Y,Z),   p(l2,X1,Y,Z),  {X>= 0,X1=X+1},    2).  % C2
r(p(l2,X,Y,Z),   p(l3,X,Y,Z),   {X<0},             3).  % C3
r(p(l3,X,Y,Z),   p(l4,X,Y,Z),   {Y=:= 25},         4).  % C4
r(p(l4,X,Y,Z),   p(l5,X,Y,Z),   {Y=\= 25},         5).  % C5
r(p(l5,X,Y,Z),   p(l6,X,Y,Z1),  {Z1=:= -1},        6).  % C6
r(p(l6,X,Y,Z),   p(l6,X,Y,Z1),  {Z=\= 0,Z1=Z-1},   7).  % C7
r(p(l6,X,Y,Z),   p(err,X,Y,Z1), {Z=:= 0},          8).  % C8

s(0, p(err,X,Y,Z),     {},    1,      (0,0)).

b(p(l1, _, _, _),  {}, 0).
b(p(l2, _, _, _),  {}, 1).
b(p(l3, _, _, _),  {}, 2).
b(p(l4, _, _, _),  {}, 3).
b(p(l5, _, _, _),  {}, 4).
b(p(l6, _, _, _),  {}, 5).
b(p(l7, _, _, _),  {}, 6).
b(p(l8, _, _, _),  {}, 7).
b(p(err, _, _, _), {}, 8).
```

A.2 inssort.c

```
:- multifile s/5.
:- multifile r/4.
```

```
:- multifile b/3.

r(p(init,K,N,I),    p(for,K1,N,I),     {K1= 1.0},  1).
r(p(for,K,N,I),     p(entryA1,K,N,I), {K=<N-1.0}, 2).
r(p(entryA1,K,N,I), p(while,K,N,I1),  {I1=K-1.0}, 3).
r(p(while,K,N,I),   p(entryA2,K,N,I), {I>= 0.0},  5).
r(p(entryA2,K,N,I), p(while,K,N,I1),  {I1=I-1.0}, 6).
r(p(while,K,N,I),   p(entryA3,K,N,I), {I=< -1.0}, 7).
r(p(entryA3,K,N,I), p(for,K1,N,I),    {K1=K+1.0}, 8).
r(p(for,K,N,I),     p(end,K,N,I),     {K>=N},     9).

s(0, p(entryA1,K,N,I), {K>=N},     1, (0,0)).
s(0, p(entryA1,K,N,I), {K=< -1.0}, 2, (0,0)).
s(0, p(entryA3,K,N,I), {I>=N-1.0}, 3, (0,0)).
s(0, p(entryA3,K,N,I), {I=< -2.0}, 4, (0,0)).
s(0, p(entryA2,K,N,I), {I=< -1.0}, 5, (0,0)).
s(0, p(entryA2,K,N,I), {I=< -2.0}, 6, (0,0)).
s(0, p(entryA2,K,N,I), {I>=N-1.0}, 7, (0,0)).
s(0, p(entryA2,K,N,I), {I>=N},     8, (0,0)).

b(p(for,_,_,_), {}, 2).
b(p(entryA1,_,_,_), {}, 3).
b(p(while,_,_,_), {}, 4).
b(p(entryA2,_,_,_), {}, 5).
b(p(entryA3,_,_,_), {}, 6).
b(p(end,_,_,_), {}, 7).
```

A.3 Coffee Machine

```
:- multifile s/5.
:- multifile r/4.
:- multifile b/3.

% Variables:
%
% C1 - first counter
% C1 - second counter
% E  - event counter
% W  - wait counter
% Paid - paid amount
% Price - coffee price

% transitions:
r(p(init, C1, C2, E, W, Paid, Price),
  p(pay, C1, C2, Ep, W, Paid, Price),
  {}, 1). % from init to pay

r(p(pay, C1, C2, E, W, Paid, Price),
```

```
  p(pay, C1, C2, Ep, W, Paidp, Price),
  {Paid < Price, Paidp =:= Paid + N, N >= 0}, 2).
% from pay to pay, if Paid < Price, Paidp = Paid + N

r(p(pay, C1, C2, E, W, Paid, Price),
  p(button, C1, C2p, Ep, Wp, Paid, Price),
  {Paid >= Price, C2p=:=0, Ep=:=0, Wp=:=0}, 3).
% from pay to button if Paid >= Price

r(p(button, C1, C2, E, W, Paid, Price),
  p(cup, C1p, C2, Ep, W, Paid, Price),
  {C2 < 15, E > 0, C1p=:=0, Ep=:=0}, 4). % from button to cup

r(p(cup, C1, C2, E, W, Paid, Price),
  p(coffee, C1, C2, Ep, W, Paid, Price),
  {C1 >= 2, C2 < 15, E > 0, Ep=:=0}, 5). % from cup to coffee

% time in the states
r(p(init, C1, C2, E, W, Paid, Price),
  p(init, C1p, C2p, Ep, Wp, Paid, Price),
  {C1p=:=C1+1, C2p=:=C2+1, Ep=:=E+1, Wp=:=W+1}, 10).
r(p(pay, C1, C2, E, W, Paid, Price),
  p(pay, C1p, C2p, Ep, Wp, Paid, Price),
  {C1p=:=C1+1, C2p=:=C2+1, Ep=:=E+1, Wp=:=W+1}, 11).
r(p(button, C1, C2, E, W, Paid, Price),
  p(button, C1p, C2p, Ep, Wp, Paid, Price),
  {C1p=:=C1+1, C2p=:=C2+1, Ep=:=E+1, Wp=:=W+1}, 12).
r(p(cup, C1, C2, E, W, Paid, Price),
  p(cup, C1p, C2p, Ep, Wp, Paid, Price),
  {C1p=:=C1+1, C2p=:=C2+1, Ep=:=E+1, Wp=:=W+1}, 13).
r(p(coffee, C1, C2, E, W, Paid, Price),
  p(coffee, C1p, C2p, Ep, Wp, Paid, Price),
  {C1p=:=C1+1, C2p=:=C2+1, Ep=:=E+1, Wp=:=W+1}, 14).

s(0, p(coffee, C1, C2, E, W, Paid, Price), {W =< 15}, 1, (0,0)).

b(p(pay,_,_,_,_,_,_), {}, 2).
b(p(button,_,_,_,_,_,_), {}, 3).
b(p(cup,_,_,_,_,_,_), {}, 4).
b(p(coffee,_,_,_,_,_,_), {}, 5).
```

A.4 Fischer's Protocol

```
:- multifile s/5.
:- multifile r/4.
:- multifile b/3.

r(p(init,A,B,K), p(aa,A1,B1,K1),  {K1= 0.0,A1= 0.0,B1= 0.0}, 0).
```

```
r(p(aa,A,B,K), p(ba,A1,B,K),  {K= 0.0,A1= 0.0}, 11).
r(p(ab,A,B,K), p(bb,A1,B,K),  {K= 0.0,A1= 0.0}, 12).
r(p(ac,A,B,K), p(bc,A1,B,K),  {K= 0.0,A1= 0.0}, 13).
r(p(as,A,B,K), p(bs,A1,B,K),  {K= 0.0,A1= 0.0}, 14).

r(p(ba,A,B,K), p(ca,A1,B,K1), {A=< 1.0,A1= 0.0,K1= 1.0}, 21).
r(p(bb,A,B,K), p(cb,A1,B,K1), {A=< 1.0,A1= 0.0,K1= 1.0}, 22).
r(p(bc,A,B,K), p(cc,A1,B,K1), {A=< 1.0,A1= 0.0,K1= 1.0}, 23).
r(p(bs,A,B,K), p(cs,A1,B,K1), {A=< 1.0,A1= 0.0,K1= 1.0}, 24).

r(p(ca,A,B,K), p(sa,A,B,K),   {A>= 2.0,K= 1.0}, 31).
r(p(cb,A,B,K), p(sb,A,B,K),   {A>= 2.0,K= 1.0}, 32).
r(p(cc,A,B,K), p(sc,A,B,K),   {A>= 2.0,K= 1.0}, 33).
r(p(cs,A,B,K), p(ss,A,B,K),   {A>= 2.0,K= 1.0}, 34).

r(p(ca,A,B,K), p(aa,A,B,K),   {K=\= 1.0}, 41).
r(p(cb,A,B,K), p(ab,A,B,K),   {K=\= 1.0}, 42).
r(p(cc,A,B,K), p(ac,A,B,K),   {K=\= 1.0}, 43).
r(p(cs,A,B,K), p(as,A,B,K),   {K=\= 1.0}, 44).

r(p(sa,A,B,K), p(aa,A,B,K1),  {K1= 0.0}, 51).
r(p(sb,A,B,K), p(ab,A,B,K1),  {K1= 0.0}, 52).
r(p(sc,A,B,K), p(ac,A,B,K1),  {K1= 0.0}, 53).
r(p(ss,A,B,K), p(as,A,B,K1),  {K1= 0.0}, 54).

r(p(aa,A,B,K), p(ab,A,B1,K),  {K= 0.0,B1= 0.0}, 61).
r(p(ba,A,B,K), p(bb,A,B1,K),  {K= 0.0,B1= 0.0}, 62).
r(p(ca,A,B,K), p(cb,A,B1,K),  {K= 0.0,B1= 0.0}, 63).
r(p(sa,A,B,K), p(sb,A,B1,K),  {K= 0.0,B1= 0.0}, 64).

r(p(ab,A,B,K), p(ac,A,B1,K1), {B =< 1.0,B1= 0.0,K1= 2.0}, 71).
r(p(bb,A,B,K), p(bc,A,B1,K1), {B =< 1.0,B1= 0.0,K1= 2.0}, 72).
r(p(cb,A,B,K), p(cc,A,B1,K1), {B =< 1.0,B1= 0.0,K1= 2.0}, 73).
r(p(sb,A,B,K), p(sc,A,B1,K1), {B =< 1.0,B1= 0.0,K1= 2.0}, 74).

r(p(ac,A,B,K), p(as,A,B,K),   {B>= 2.0,K= 2.0}, 81).
r(p(bc,A,B,K), p(bs,A,B,K),   {B>= 2.0,K= 2.0}, 82).
r(p(cc,A,B,K), p(cs,A,B,K),   {B>= 2.0,K= 2.0}, 83).
r(p(sc,A,B,K), p(ss,A,B,K),   {B>= 2.0,K= 2.0}, 84).

r(p(ac,A,B,K), p(aa,A,B,K),   {K=\= 2.0}, 91).
r(p(bc,A,B,K), p(ba,A,B,K),   {K=\= 2.0}, 92).
r(p(cc,A,B,K), p(ca,A,B,K),   {K=\= 2.0}, 93).
r(p(sc,A,B,K), p(sa,A,B,K),   {K=\= 2.0}, 94).

r(p(as,A,B,K), p(aa,A,B,K1),  {K1= 0.0},101).
r(p(bs,A,B,K), p(ba,A,B,K1),  {K1= 0.0},102).
r(p(cs,A,B,K), p(ca,A,B,K1),  {K1= 0.0},103).
```

```
r(p(ss,A,B,K), p(sa,A,B,K1),  {K1= 0.0},104).

s(0, p(ss,A,B,K), {}, 1, (0,0)).

b(p(aa,_,_,_), {}, 2).
b(p(ba,_,_,_), {}, 3).
b(p(ab,_,_,_), {}, 4).
b(p(bb,_,_,_), {}, 5).
b(p(ac,_,_,_), {}, 6).
b(p(bc,_,_,_), {}, 7).
b(p(as,_,_,_), {}, 8).
b(p(bs,_,_,_), {}, 9).
b(p(ca,_,_,_), {}, 10).
b(p(cb,_,_,_), {}, 11).
b(p(cc,_,_,_), {}, 12).
b(p(cs,_,_,_), {}, 13).
b(p(sa,_,_,_), {}, 14).
b(p(sb,_,_,_), {}, 15).
b(p(sc,_,_,_), {}, 16).
b(p(ss,_,_,_), {}, 17).
```