

Abstractions in Model Checking

B. Tech. Seminar Report

Submitted in partial fulfillment of the requirements
for the degree of

Bachelor-of-Technology

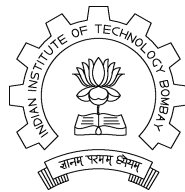
by

Varun Kanade

Roll No: 02005021

under the guidance of

Prof. Supratik Chakraborty



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Abstract

We consider the use of abstraction techniques in model checking. First we discuss a general framework for talking about abstractions, as proposed by P. Cousot and R. Cousot. We then study use of abstractions in symbolic model checking using **ACTL*** and refinement of abstractions using counterexamples as proposed by Clarke et. al. This allows us to verify very large finite state systems. We then consider predicate abstraction, which combines theorem proving and model checking. Here also we talk about refinement of abstractions using counterexamples, and find successive approximations to the abstract transition relations as discussed by S. Das and D.Dill.

1 Introduction

Formal Verification in Computer Science refers to the act of proving or disproving the correctness of a (software or hardware) system with respect to certain formal specification or property, using formal methods. Such verification is necessary for safety-critical systems. Model Checking [CGL94b] is an automatic technique for verifying finite-state reactive systems (e.g. sequential circuit designs, communication protocols). Section 2 begins with a few mathematical definitions and theorems which we will require subsequently. Section 3 briefly describes what model checking means. The specification is given in the form of temporal logic formulas (e.g. **CTL**, **CTL***, **ACTL**, **ACTL***). The transition relation is expressed as a state-transition graph. Model checking has been successfully applied to small finite-state systems, however it cannot directly be applied to large systems. Ordered Binary Decision Diagrams (OBDDs) [Bry92] provide a convenient way of representing boolean functions. Section 4 gives a brief description of OBDDs and some results. Using OBDDs in model checking allows us to obtain efficient algorithms. This method is known as symbolic model checking. It allows much larger systems to be verified. Symbolic model checking is used to verify sequential circuit designs with up to 10^{20} states in [BCL⁺94]. However even symbolic model checking does not allow very large systems and systems with infinite states to be verified. For verifying such systems, we need to use abstraction techniques. Section 6 describes abstract interpretations of programs in a lattice-theoretic framework and levels of abstraction as Galois connections [CC77]. This gives a framework for using abstractions in program analysis. The next section uses abstraction to reduce the size of the state space and then apply model checking to it [CGL94a]. However the abstraction function that we use may not be the best possible. Section 8 tries to find an initial abstraction function and then if necessary refine the abstraction functions using counter-examples generated [CGJ⁺00]. These methods are useful in handling very large finite-state systems. In Section 9 predicate abstraction is discussed. Predicate abstraction can be used for verification of systems with infinite state spaces [DDP99]. Predicate abstraction combines the power of theorem proving and model checking. Section 10 concludes with possible further developments.

2 Some Mathematical Preliminaries

We assume that the reader is familiar with temporal logics such as **CTL**, **CTL***, **ACTL**, **ACTL***. Some details of these are provided in [CGL94b, CGL94a]. We now define what is meant by lattice and a monotone function.

Definition 2.1 A poset $\mathcal{L} = \langle L, \leq \rangle$ is said to be a **complete lattice** if $\forall S \subseteq L, \bigvee_L S$ and $\bigwedge_L S$ exist in L .

Definition 2.2 A function f from a poset \mathcal{A} to a poset \mathcal{B} is said to be *monotone* if $\forall x, y \in \mathcal{A}, x \leq y \Rightarrow f(x) \leq f(y)$.

The next theorem talks about the fixed points of a monotone function on a complete lattice. The theorem is due to Alfred Tarski. Tarski's original paper [Tar55] has a very elegant proof. A constructive proof for the same theorem is given in [Ech].

Theorem 2.1 Tarski's Fixed-Point Theorem

Let $\mathcal{L} = \langle L, \leq \rangle$ be a lattice and $f : L \rightarrow L$ be a monotone function. Then

1. Let $P = \{x \in L \mid f(x) = x\}$ denote the fixed-point set of f . Then $\mathcal{P} = \langle P, \leq \rangle$ is a non-empty complete lattice.
2. The least fixed-point $\bigwedge_L P = \bigwedge_L \{x \in L \mid f(x) \leq x\} \in P$
3. The greatest fixed-point $\bigvee_L P = \bigvee_L \{x \in L \mid f(x) \geq x\} \in P$

This theorem will be useful in Section 5 which discusses symbolic model checking by computing least fixed-points. Also in section 6 we will use it to guarantee existence of fixpoints in the framework discussed.

We now go on to define the model checking problem in the next section.

3 Model Checking

For model checking we use the notion of a Kripke structure with fairness constraints. A Kripke structure may be defined as a tuple

$$M = \langle S, S_0, AP, L, R, F \rangle, \text{ where}$$

1. S is a finite set of states and $S_0 \subseteq S$ is the set of start states
2. AP is a finite set of atomic propositions
3. $L : S \rightarrow \mathcal{P}(AP)$ is the labeling function which gives which atomic propositions are true in each state
4. $R \subseteq S \times S$ is the transition relation
5. $F \subseteq \mathcal{P}(S)$ are fairness constraints given as Büchi acceptance condition

A path in a Kripke structure M is a sequence, $\pi = s_0 s_1 s_2 \dots$, such that $\forall i \geq 0, R(s_i, s_{i+1})$. Define $inf(\pi)$ as $inf(\pi) = \{s \mid s = s_i \text{ for infinitely many } i\}$. A path π is said to be fair if $\forall P \in F, inf(\pi) \cap P \neq \emptyset$. π^n denotes path from s_n onwards.

If a state formula f holds in state s of M , we denote it by $M, s \models f$. If f holds along a path π in M , we denote it by $M, \pi \models f$. We write $M \models f$ if f holds in all initial states of M .

The model checking problem is given a Kripke structure M , determine whether a formula ϕ holds i.e. $M \models \phi$, and if not then produce a counter-example. The formula ϕ may be given in logic systems such as **CTL**, **ACTL**, **CTL*** or **ACTL***.

4 Study of Ordered Binary Decision Diagrams

Ordered Binary Decision Diagrams (OBDDs) are similar to Binary Decision Diagrams, however instead of trees, they are represented as directed acyclic graphs(DAGs). Also in OBDDs we require a strict total ordering on the boolean variables in the formula. A good description of OBDDs and algorithms manipulating them is given in [Bry92]. In this paper Bryant proves that given a variable ordering, a unique canonical forms exist. This is useful, because the check for equivalence between two boolean formulas is eliminated. However the size of the graph may depend heavily on the ordering of variables chosen. Figure 1 illustrates how different orderings may lead to different sizes of graphs. However human understanding of the problem and

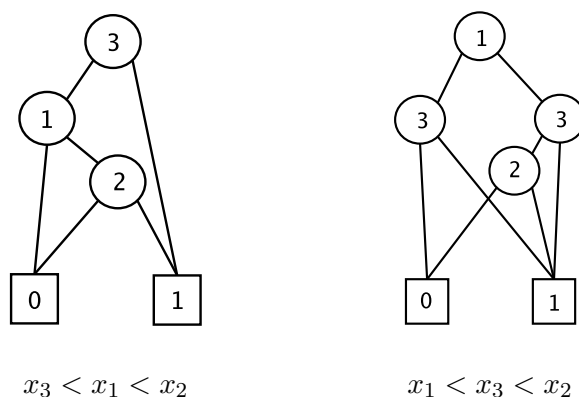


Figure 1: Ordering affects size of OBDD

some heuristics are good enough to reduce graph sizes for most commonly encountered boolean formulas. A boolean formula is represented as a function $f(v_1, v_2, \dots, v_n)$ where v_1, v_2, \dots, v_n are boolean variables. $f|_{x_i=b}(v_1, v_2, \dots, v_n)$ denotes restriction of $f(v_1, v_2, \dots, v_n)$ where value of v_i is restricted to $b \in \{0, 1\}$. This is useful to write formulas involving quantifiers on boolean variables,

$$\begin{aligned} \exists x_i f &= f|_{x_i=0} \vee f|_{x_i=1} \\ \forall x_i f &= f|_{x_i=0} \wedge f|_{x_i=1} \end{aligned}$$

The table below shows time complexity for all the operations required for manipulations of boolean formulas.

Procedure	Result	Time Complexity
Reduce	G reduced to canonical form	$O(G .log G)$
Apply	$f_1 \langle op \rangle f_2$	$O(G_1 . G_2)$
Restrict	$f _{x_i=b}$	$O(G .log G)$
Compose	$f_1 _{x_i=f_2}$	$O(G_1 ^2. G_2)$

5 Symbolic Model Checking

Model checking means determining whether a given formula f is satisfied in a state given a transition relation R . If states are represented explicitly, as the problem size grows large, the list or table to store grows in proportion to number of states. To avoid this state explosion problem, we represent the state space symbolically rather than explicitly. This is known as

symbolic model checking. One such method to represent states symbolically is using OBDDs discussed in the previous section. Transition relation $R(\bar{v}_i, \bar{v}_f)$ and f are also represented as OBDDs.

We use a function $BDD(f, R)$ that takes the transition relation and a formula f , and returns a BDD that has the property that $BDD(f, R)(v)$ is true iff the formula f is true in state v under the transition relation R . $BDD(\mathbf{EX}f, R)(\bar{v}_i) \equiv \exists \bar{v}_f [R(\bar{v}_i, \bar{v}_f) \wedge BDD(f, R)(\bar{v}_f)]$

$\mathbf{EFU}g$ is the least fixpoint of $Z = g \vee [f \wedge \mathbf{EX}Z]$.

$\mathbf{EG}f$ is the greatest fixpoint of $Z = f \wedge \mathbf{EX}Z$.

We can consider $Pred(S)$ as set of predicates, which may be considered as lattice of subsets of S , the state space. Each predicate corresponds to a subset of S on which the predicate is true. This guarantees by Tarski's theorem the existence of least and greatest fixpoints given above and also characterizes them. We can then use iterative methods for computation of BDD representations of these fixpoints, using BDD representations of f and g . Other **CTL** formulas can be expressed in terms of these three. Thus given f , we can recursively apply these rules for model checking. The algorithms for doing all these manipulations of BDDs is given in [Bry92].

6 Abstract Interpretations - Lattice and Fixed-Point Theory

In this section we slightly divert from model checking, and look at abstractions in a more general framework of program analysis as discussed by P. Cousot and R. Cousot in [CC77]. In this we talk about **contexts** which may in general be any lattice. The interpretation of the program is defined by a function **Int**, which takes the contexts at a program point and returns the context at the next program point. We can talk about levels of abstractions depending on the contexts used and an abstraction function α that maps a set of contexts onto another. The idea of abstraction and symbolic execution introduced here are useful in studying abstractions in model checking.

6.1 Semantics of Programs

A program is considered as a flowchart with nodes and arcs. The nodes may be **Entries**, **Assignments**, **Tests**, **Junctions**, **Exits**. An arc goes from one node to another called **origin** and **end** of arc respectively. Let **Ident** and **Values** be the set of identifiers and values. Let **Expr**, **BExpr** \subseteq **Expr** and **Env** = **Ident** \rightarrow **Values** denote the set of expressions, boolean expressions and environments respectively. A state of the program is defined as a pair $\langle \mathbf{arc}, \mathbf{env} \rangle$. Let **I-state** be the initial states. The function **n-state** defined in Algorithm 1 gives the next state, in a computation. The functions **a-succ**(n) and **a-pred**(n) denote successor and predecessor of the node n respectively.

For a program we may define contexts in terms of environment associated with arcs. If q is an arc in a program then define the context associated with q , as follows

$$C_q = \{e \mid \exists n \geq 0, \exists i_s \in \mathbf{I-states} \mid \langle q, e \rangle = \mathbf{n-state}^n(i_s)\}$$

Also we can talk about a context vector C_v , which will associate a context with each arc of the program. As we have seen, a context is a subset of environments. The function **n-context** gives a new context at an arc given a current context vector C_v . We define a function **F-Cont**(C_v) = $\lambda r. \mathbf{n-context}(r, C_v)$, which gives a new context vector from a given one. This function is a monotonic function, and using Tarski's Theorem we can say that this will have fixpoints, which will give the contexts to be associated with each arc.

Algorithm 1 $\mathbf{n\text{-state}}(s)$

$n := \text{end}(\text{arc}(s)), e = \text{env}(s)$
if n in **Assignments** **then**
 return $\langle \mathbf{a\text{-succ}}(n), e[\text{val}[\llbracket \text{expr}(n) \rrbracket](e)/\text{id}(n)] \rangle$
else if n in **Tests** **then**
 return $\langle \text{cond}(\text{val}[\llbracket \text{test}(n) \rrbracket](e), \mathbf{a\text{-succ-t}}(n), \mathbf{a\text{-succ-f}}(n)), e \rangle$
else if n in **Junctions** **then**
 return $\langle \mathbf{a\text{-succ}}(n), e \rangle$
else if n in **Exits** **then**
 return s
end if

Algorithm 2 $\mathbf{n\text{-context}}(q, C_v)$

$n = \text{origin}(r)$
if n in **Entries** **then**
 return $\{\lambda x. \perp_{\text{Env}}\}$
else if n in **Assignments, Tests, Junctions** **then**
 return $\bigcup_{q \in \mathbf{a\text{-pred}}(n)} \bigcup_{e \in C_v(q)} \text{cond}(\text{arc}(s = \mathbf{n\text{-state}}(\langle q, e \rangle)) = r, \text{env}(s), \emptyset)$
end if

6.2 Abstract Interpretations of Programs

An abstract interpretation I of a program is a tuple

$$I = \langle \mathbf{A\text{-Cont}}, \circ, \leq, \top, \perp, \mathbf{Int} \rangle$$

Here $\mathbf{A\text{-Cont}}$ is a complete \circ -semilattice with ordering \leq , such that $x \leq y \Leftrightarrow x \circ y = y$. \top and \perp are the top and bottom elements of the $\mathbf{A\text{-Cont}}$ respectively. This in fact makes $\mathbf{A\text{-Cont}}$ a complete lattice. Let $\mathbf{V\text{-Cont}}$ be the set of context vectors $\mathbf{V\text{-Cont}} = \mathbf{Arcs} \rightarrow \mathbf{A\text{-Cont}}$. $\mathbf{Int} : \mathbf{Arcs} \times \mathbf{V\text{-Cont}} \rightarrow \mathbf{A\text{-Cont}}$ defines the interpretations of the program. So given the input contexts at a point, it gives the output context. So we can define a function \mathbf{FInt} as

$$\mathbf{FInt} : \mathbf{V\text{-Cont}} \rightarrow \mathbf{V\text{-Cont}} \mid \mathbf{FInt}(C_v) = \lambda r. \mathbf{Int}(f, C_v)$$

We can extend definitions of \circ and \leq for context vectors, so $\mathbf{V\text{-Cont}}$ becomes a complete lattice and the function \mathbf{FInt} is monotonic, and hence has fixpoints.

6.3 Consistent Abstract Interpretations

An abstract interpretation $I' = \langle \mathbf{A\text{-Cont}'}, \circ', \leq', \top', \perp', \mathbf{Int}' \rangle$ of a program is said to be consistent to the concrete interpretation $I = \langle \mathbf{C\text{-Cont}}, \circ, \leq, \top, \perp, \mathbf{Int} \rangle$ if the following hold

1. $\alpha : \mathbf{C\text{-Cont}} \rightarrow \mathbf{A\text{-Cont}'}, \gamma : \mathbf{A\text{-Cont}'} \rightarrow \mathbf{C\text{-Cont}}$
2. α and γ are order-preserving
3. $\forall x' \in \mathbf{A\text{-Cont}'}, x' = \alpha(\gamma(x'))$
4. $\forall x \in \mathbf{C\text{-Cont}}, x \leq \gamma(\alpha(x))$
5. If C'_v and C_v are the context vectors which are fixpoints of \mathbf{FInt}' and \mathbf{FInt} respectively, then $C_v \leq \tilde{\gamma}(C'_v)$ and $\alpha(\tilde{C}_v) \leq C'_v$, where $\tilde{\alpha}(C) = \lambda r. \alpha(C(r))$ and $\tilde{\gamma}(C') = \lambda r. \gamma(C'(r))$.

We then say that $I \leq (\alpha, \gamma)I'$ and I and I' are said to have a Galois connection. This relation between abstract interpretations forms a partial order. This is because we can show that if $I \leq (\alpha, \gamma)I'$ and $I' \leq (\alpha', \gamma')I$, then the algebras I and I' are in fact isomorphic. In this manner we get a lattice of abstract interpretations in which I_{SS} , called the static semantics of the program and is discussed in the next section is the bottom. This is considered as the most concrete interpretation of the program. $\top_I = \langle \{\mathbf{I}\}, \lambda(x, y).y, \lambda(x, y).\mathbf{true}, \mathbf{I}, \mathbf{I}, \lambda(a, C).\mathbf{I} \rangle$ is the top, which is the most abstract interpretation of the program.

6.4 Static and Deductive Semantics

We defined earlier contexts as subset of \mathbf{Env} . This is called static semantics of the program. We can see that under the inclusion relation \subseteq the set of contexts forms a complete lattice. So we get an abstract interpretation of the program as

$$I_{SS} = \langle \mathcal{P}(\mathbf{Env}), \cup, \subseteq, \mathbf{Env}, \emptyset, \mathbf{n-context} \rangle$$

Also let \mathbf{Pred} denote set of predicates over the program variables of the form $P(x_1, \dots, x_n)$ where $(x_1, \dots, x_n) \in \mathbf{Ident}^n$, then the following is also an interpretation the program,

$$I_{DS} = \langle \mathbf{Pred}, \mathbf{or}, \Rightarrow, \mathbf{true}, \mathbf{false}, \mathbf{n-pred} \rangle$$

where $\mathbf{n-pred}$ is defined in algorithm 3

Define α and γ as

Algorithm 3 $\mathbf{n-pred}(r, P_v)$

```

 $n = \mathbf{origin}(r), p = \mathbf{a-pred}(n)$ 
if  $n$  in  $\mathbf{Entries}$  then
  return  $(\forall x, x = \perp_{\mathbf{Values}})$ 
else if  $n$  in  $\mathbf{Junctions}$  then
  return  $\mathbf{or}_{q \in p}(P_v(q))$ 
else if  $n$  in  $\mathbf{Tests}$  then
  return  $P_v(p)$  and  $\mathbf{cond}(r = \mathbf{a-succ-t}(n), \mathbf{test}(n), \neg \mathbf{test}(n))$ 
else if  $n$  in  $\mathbf{Assignments}$  then
   $P = P_v(p), x = \mathbf{id}(n), e = \mathbf{expr}(n)$ 
  return  $\exists v, (P[v/x] \mathbf{and} x = e[v/x])$ 
end if

```

$\alpha : \mathcal{P}(\mathbf{Env}) \rightarrow \mathbf{Pred}, \alpha(C) = (\mathbf{or}_{e \in C}(\mathbf{and}_{x \in \mathbf{Ident}}(x = e(x))))$

$\gamma : \mathbf{Pred} \rightarrow \mathcal{P}(\mathbf{Env}), \gamma(P) = e | P(e(x)/x, x \in \mathbf{Ident})$

Then we can show that $I_{SS} \leq (\alpha, \gamma)I_{DS}$.

In the next section, we now look at abstractions in model checking, and we will see that this falls into the general ideas discussed in this section.

7 Using Abstractions in Model Checking

Modern systems are very complex and would contain very large number of states. Verification of such systems is not possible using symbolic model checking alone. In [BCL⁺94] using symbolic model checking, a system with around 10^{20} states was verified. In some further work, verification of systems with up to 10^{100} states was achieved using symbolic model checking. Combining abstractions with model checking leads to much better results. In [CGL94a] verification of a pipelined ALU with 64 registers, each 64 bits wide, is discussed and the system contains about

10^{1300} states.

While using abstractions, we try to obtain approximate abstract systems by symbolic execution. For formulas in **ACTL***, the abstraction is conservative. When abstraction functions are congruences modulo the primitive operators used, the abstraction is exact.

7.1 Transition Systems and Abstractions

We would consider systems with only a finite set of variables say v_1, \dots, v_n , where the domain of v_i is D_i and is finite. We define set of states as $D = D_1 \times \dots \times D_n$. A transition system is defined as a tuple $M = \langle S, I, R \rangle$ where S is the set of states which is D . $I \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is the transition relation.

Let \hat{D}_i denote abstract domain corresponding to the concrete domain D_i and let $h_i : D_i \rightarrow \hat{D}_i$ be a surjection which is the abstraction function. Let $\hat{D} = \hat{D}_1 \times \dots \times \hat{D}_n$. Let $h : D \rightarrow \hat{D}$ be the complete abstraction function defined as $h(d_1, \dots, d_n) = (h_1(d_1), \dots, h_n(d_n))$.

Alternately the abstraction functions may be viewed as equivalence relations on the domain D_i , $\equiv_i \subseteq D_i \times D_i$, where $d_i \equiv_i e_i \Leftrightarrow h_i(d_i) = h_i(e_i)$. This induces an equivalence relation on D , where $(d_1, d_2, \dots, d_n) \equiv (e_1, e_2, \dots, e_n) \Leftrightarrow \bigwedge_i d_i \equiv_i e_i$.

We now want to construct abstract transition systems $\hat{M} = \langle \hat{D}, \hat{I}, \hat{R} \rangle$. We say that \hat{M} approximates M , denoted by $M \sqsubseteq_h \hat{M}$ when

1. $\exists d(h(d) = \hat{d} \wedge I(d)) \Rightarrow \hat{I}(\hat{d})$
2. $\exists d_1, d_2(h(d_1) = \hat{d}_1 \wedge h(d_2) = \hat{d}_2 \wedge R(d_1, d_2)) \Rightarrow \hat{R}(\hat{d}_1, \hat{d}_2)$

We can define a minimal abstract transition state $\hat{M}_{min} = \langle \hat{D}, \hat{I}_{min}, \hat{R}_{min} \rangle$ as

1. $\hat{I}_{min}(\hat{d})$ iff $\exists d(h(d) = \hat{d} \wedge I(d))$
2. $\hat{R}_{min}(\hat{d}_1, \hat{d}_2)$ iff $\exists d_1 \exists d_2(h(d_1) = \hat{d}_1 \wedge h(d_2) = \hat{d}_2 \wedge R(d_1, d_2))$

\hat{M}_{min} is the most accurate abstraction of the system, however given the symbolic representation (OBDDs) of the relations I and R , it is not efficient to compute the relations \hat{I}_{min} and \hat{R}_{min} . For this we try to find an approximation \hat{M}_{app} to the abstract transition system \hat{M}_{min} .

7.2 Producing Abstract Models

Given a program, using semantics of the language we can generate relational expressions \mathcal{I} and \mathcal{R} , which are formulas in first-order predicate logic which will be built using *primitive relations* for the basic operators and constants in the language. Using this we can obtain the initial states I and the transition relation R . Below we give an example (Algorithm 4), to show how this can be obtained from the program code. This program given n , computes the product of first n numbers. The relational expressions for \mathcal{I} and \mathcal{R} are as follows:

Algorithm 4 Product of n numbers

```

p = 1
while n ≠ 0 do
  p = p * n;
  n = n - 1;
end while

```

$$\begin{aligned}
\mathcal{R} &= (PC = 0 \wedge p' = 1 \wedge n' = n \wedge PC' = 1) \\
&\quad \vee (PC = 1 \wedge p' = p * n \wedge n' = n - 1 \wedge PC' = 1) \\
&\quad \vee (PC = 1 \wedge n = 0 \wedge p' = p \wedge n' = n \wedge PC' = 2) \\
&\quad \vee (PC = 2 \wedge p' = p \wedge n' = n \wedge PC' = 2) \\
\mathcal{I} &= (PC = 0 \wedge p' = 1 \wedge n' = n \wedge PC' = 0)
\end{aligned}$$

After obtaining \mathcal{I} and \mathcal{R} , it is not efficient to obtain $\hat{\mathcal{I}}_{min}$ and $\hat{\mathcal{R}}_{min}$. We show how to obtain $\hat{\mathcal{I}}_{app}$ and $\hat{\mathcal{R}}_{app}$.

Let ϕ, ϕ_1, ϕ_2 denote relational expressions built up from the primitive relations representing the operations of the program. Let

$$[\phi](\hat{x}_1, \dots, \hat{x}_n) = \exists x_1 \dots \exists x_n (\bigwedge_i (h_i(x_i) = \hat{x}_i) \wedge \phi(x_1, \dots, x_n)).$$

Clearly we can see that $\hat{\mathcal{I}}_{min} = [\mathcal{I}]$ and $\hat{\mathcal{R}}_{min} = [\mathcal{R}]$.

Define a transformation function \mathcal{T} as follows

1. If P is a primitive relation, $\mathcal{T}(P(x_1, x_2, \dots, x_n)) = [P](\hat{x}_1, \hat{x}_2, \dots, \hat{x}_n)$ and $\mathcal{T}(\neg P(x_1, x_2, \dots, x_n)) = [\neg P](\hat{x}_1, \hat{x}_2, \dots, \hat{x}_n)$
2. $\mathcal{T}(\phi_1 \wedge \phi_2) = \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2)$
3. $\mathcal{T}(\phi_1 \vee \phi_2) = \mathcal{T}(\phi_1) \vee \mathcal{T}(\phi_2)$
4. $\mathcal{T}(\mathbf{A}x\phi) = \mathbf{A}\hat{x}\mathcal{T}(\phi)$
5. $\mathcal{T}(\mathbf{E}x\phi) = \mathbf{E}\hat{x}\mathcal{T}(\phi)$

From the above definition we can see that \mathcal{T} applies the operation $[\cdot]$ only at the innermost level. In 2 and 4 above, \mathcal{T} pushes existential quantifiers over conjunctions, which would make the two formulas inequivalent. We have seen this in Section 6 regarding deductive semantics of a program as an abstract interpretation. The static semantics of the program can be viewed as the most concrete interpretation, and the deductive semantics is an abstraction of it. In the discussion above the abstraction function h corresponds to α and h^{-1} to γ as defined in section 6.3. Here the abstraction is applied directly only to the innermost level, that is that of primitive operators. This results in an approximation.

We now state a theorem from [CGL94a]

Theorem 7.1 $[\phi]$ implies $\mathcal{T}(\phi)$. In particular, $[\mathcal{I}]$ implies $\mathcal{T}(\mathcal{I})$, and $[\mathcal{R}]$ implies $\mathcal{T}(\mathcal{R})$. Hence we get that $M \sqsubseteq_h \hat{M}_{app}$.

Definition 7.1 If $P(x_1, \dots, x_m)$ is a relation with x_i ranging over D_{i_j} . The equivalence relations \equiv_{i_j} are a congruence with respect to P if

$$\forall d_1 \dots \forall d_m \forall e_1 \dots \forall e_m (\bigwedge_j d_j \equiv_{i_j} e_j \rightarrow (P(d_1, \dots, d_m) \Leftrightarrow P(e_1, \dots, e_m)))$$

Let \hat{M} be a transition system over \hat{D} . We say that \hat{M} exactly approximates M (denoted by $M \approx_h \hat{M}$) when $M \sqsubseteq_h \hat{M}$ and

1. $\hat{I}(\hat{d})$ implies $\forall d (h(d) = \hat{d} \rightarrow I(d))$
2. $\hat{R}(\hat{d}_1, \hat{d}_2)$ implies $\forall d_1 \forall d_2 (h(d_1) = \hat{d}_1 \wedge h(d_2) = \hat{d}_2 \rightarrow R(d_1, d_2))$

Theorem 7.2 If \equiv is a congruence with respect to the primitive relations, then $M \approx_h \hat{M}_{app}$.

7.3 ACTL* Model Checking

We have seen in the previous section, a method to generate approximate abstract transition systems for a transition system M . In general it is difficult to obtain the exact abstraction function \hat{M}_{min} . However we have seen that for **ACTL*** formulas, the approximate transition system obtained is conservative. We now define a translation \mathcal{C} between formulas in the abstract transition system \hat{M} and the concrete transition system M . The mapping is defined as follows

1. $\mathcal{C}(true) = true$, $\mathcal{C}(false) = false$, $\mathcal{C}(\hat{v}_i = \hat{d}_i) = \bigvee \{v_i = d_i \mid h_i(d_i) = \hat{d}_i\}$. $\mathcal{C}(\hat{v}_i \neq \hat{d}_i) = \neg \mathcal{C}(\hat{v}_i = \hat{d}_i)$
2. If ϕ and ψ are state formulas, then $\mathcal{C}(\phi \wedge \psi) = \mathcal{C}(\phi) \wedge \mathcal{C}(\psi)$, and $\mathcal{C}(\phi \vee \psi) = \mathcal{C}(\phi) \vee \mathcal{C}(\psi)$.
3. If ϕ is a path formula, then $\mathcal{C}(\mathbf{A}\phi) = \mathbf{A}(\mathcal{C}(\phi))$, and $\mathcal{C}(\mathbf{E}\phi) = \mathbf{E}(\mathcal{C}(\phi))$.
4. If ϕ is a path formula that is also a state formula, then $\mathcal{C}(\phi)$ is given by the above rules.
5. If ϕ and ψ are path formulas, then $\mathcal{C}(\phi \wedge \psi) = \mathcal{C}(\phi) \wedge \mathcal{C}(\psi)$, and $\mathcal{C}(\phi \vee \psi) = \mathcal{C}(\phi) \vee \mathcal{C}(\psi)$.
6. If ϕ and ψ are path formulas, then
 - (a) $\mathcal{C}(\mathbf{X}\phi) = \mathbf{X}\mathcal{C}(\phi)$,
 - (b) $\mathcal{C}(\phi \mathbf{U}\psi) = \mathcal{C}(\phi) \mathbf{U}\mathcal{C}(\psi)$,
 - (c) $\mathcal{C}(\phi \mathbf{V}\psi) = \mathcal{C}(\phi) \mathbf{V}\mathcal{C}(\psi)$.

It is easy to see that if π is a path in M , then $h(\pi)$ is a path in \hat{M} .

Theorem 7.3 *Let $M \sqsubseteq_h \hat{M}$ and let ϕ be a **ACTL*** formula describing \hat{M} . Then $\hat{M} \models \phi$ implies $M \models \mathcal{C}(\phi)$. If $M \approx_h \hat{M}$, then we have $\hat{M} \models \phi$ iff $M \models \mathcal{C}(\phi)$.*

7.4 Example

We want to argue that for $n \geq 2$ Algorithm 4, will always have value of p even at the end of execution. Figure 2 shows the abstraction function and the abstract transition system. The property is easily verified.

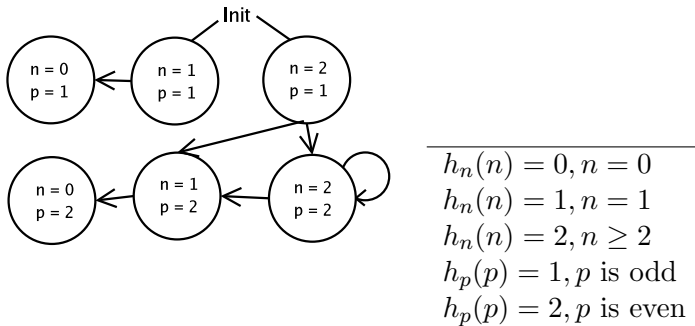


Figure 2: Abstraction of program in algorithm 4

8 Refining Abstractions Using Counter-Examples

In the previous section, we have seen how abstractions may be used in model checking very large finite-state systems. Abstraction based model checking is conservative, that is if the formula evaluates to true in the abstract transition system, then it would be true in the concrete transition system. However if it is not true in the abstract transition system, the counter-example generated may not truly be a counter-example (called spurious counter-example) in the concrete transition system. In [CGJ⁺00] a method is given to refine the original abstraction function, which will help in either proving the formula correctly, or generate a concrete counterexample. The refinement of the abstraction function should be as coarse as possible, to keep the size of the abstract transition system small. However the problem of finding the coarsest refinement, is **NP**-hard. Here the abstraction function will also be generated automatically, rather than being required to be provided manually.

8.1 Understanding the Problem

As before the program variables are assumed to be v_1, \dots, v_n with domain of v_i to be D_i . For every variable in the program a transition block B_i is defined as shown below. At any stage, if C_i^j is the first condition to be true, then the variable is assigned the value A_i^j . Atomic formulas are made up of expressions and relational symbols. Predicates are formed from atomic formulae using negation(\neg), conjunction(\wedge) and disjunction(\vee). If P is a predicate, denote by $Atoms(P)$ the set of atomic predicates of P . We say a state $d = (d_1, \dots, d_n) \models P$, if by replacing all occurrences of v_i by d_i in P , it evaluates to true. Denote by $Atoms(B_i)$ atomic propositions of the transition block B_i , and $Atoms(P)$ as the atomic propositions of the program P .

```

init( $v_i$ )   :=    $I_i$ ;
next( $v_i$ )   :=   case
                 $C_i^1$  :  $A_i^1$ ;
                 $C_i^2$  :  $A_i^2$ ;
                 $\dots$  :  $\dots$ ;
                 $C_i^k$  :  $A_i^k$ ;
esac;
                 $B_i$ 

```

The program P is modeled using a Kripke Structure $M = \langle S, I, R, L \rangle$, where $S = D$ is the set of states, $I \subseteq S$ is set of initial states, $R \subseteq S \times S$ be the transition relation and $L : S \rightarrow Atoms(P)$ the labeling function, where $L(d) = \{f \in Atoms(P) | d \models f\}$.

8.2 Generating an Initial Abstraction

Let for a given atomic formula f , $var(f)$ denote the variables occurring in f . Given a set of atomic formulas U , $var(U) = \bigcup_{f \in U} var(f)$. For e.g. $var(x = y) = \{x, y\}$. Two atomic formulas f_1 and f_2 are said to interfere iff $var(f_1) \cap var(f_2) \neq \emptyset$. Let \equiv_I denote the equivalence relation on $Atoms(P)$ obtained by taking the reflexive, transitive closure of the interference relation. We can see that each equivalence class $[f]$ here corresponds to a formula cluster FC_i and also induces an equivalence relation on the variables, $v_i \equiv_V v_j$ iff v_i and v_j occur in some atomic formulae belonging to the same formula cluster. Call the corresponding variable clusters VC_i . The initial abstraction function is constructed from these variable clusters. For a given vari-

able cluster VC_i , the domain under consideration is $D_{VC_i} = \prod_{v_j \in VC_i} D_j$. We define h_i as follows, $h_i(d_1, \dots, d_k) = h_i(e_1, \dots, e_k)$ iff for all atomic formulae $f \in FC_i$, $(d_1, \dots, d_k) \models f \Leftrightarrow (e_1, \dots, e_k) \models f$. This gives an abstraction function. The example below will make the process clear.

Example: Consider a program with variables x, y, r . Let $Atoms(P) = \{r = \mathbf{true}, x = 1, x \geq y, y = x\}$. Then the variable clusters will be formed as $VC_1 = (x, y)$, and $VC_2 = (r)$. Also let $x, y \in 0, 1, 2$ and $r \in \{\mathbf{true}, \mathbf{false}\}$. The equivalence classes corresponding to \equiv_1 will be as follows $0 = \{(0,1), (0,2)\}$, $1 = \{(1,2)\}$, $2 = \{(2,0)\}$, $3 = \{(1,0), (2,1)\}$, $4 = \{(0,0), (2,2)\}$, $5 = \{(1,1)\}$ Now we can easily obtain the abstract system.

8.3 Model Checking the Abstract Model

We know that when we try to verify the formula ϕ in the abstract model, if it is true, then it is true in the concrete model as well. However there may be counterexamples generated by the model checker on the abstract system. The counterexamples may be spurious or may be valid concrete counter examples. Here only counterexamples which are paths or loops are considered.

8.3.1 Path counterexamples

Suppose the model checker gives a path counterexample. Let this path be $\hat{T} = \langle \hat{s}_1 \dots, \hat{s}_n \rangle$. We define the concrete paths corresponding to this as $h^{-1}(\hat{T}) = \{ \langle s_1, \dots, s_n \rangle \mid \bigwedge_{i=1}^n h(s_i) = \hat{s}_i \wedge I(s_1) \wedge \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1}) \}$. The symbolic algorithm **SplitPATH** to compute $h^{-1}(\hat{T})$ is given below stating from $S_1 = h^{-1}(\hat{s}_1) \cap I$, and $S_i = \text{imag}(S_{i-1}, R) \cap h^{-1}(\hat{s}_i)$, where $\text{Imag}(S_{i-1}, R)$ denotes concrete states reachable from some state in S_{i-1} under the transition relation R . If the algorithm terminates with $S_n \neq \emptyset$, then we have a counterexample, otherwise it returns a set S and an index j , where S_j is the first empty set along the path.

Algorithm 5 SplitPATH(\hat{T})

```

 $S := h^{-1}(\hat{s}_1) \cap I$ 
 $j = 1$ 
while  $S \neq \emptyset \wedge j < n$  do
   $j = j + 1$ 
   $S_{prev} = S$ 
   $S = \text{Imag}(S, R) \cap h^{-1}(\hat{s}_j)$ 
end while
if  $S \neq \emptyset$  then
  output counterexample
else
  output  $j, S_{prev}$ 
end if

```

8.3.2 Loop counterexample

The model checker may also return a loop counterexample. It is slightly trickier to argue in this case. Let the loop counterexample be $\hat{T} = \langle \hat{s}_1, \dots, \hat{s}_i \rangle \langle \hat{s}_{i+1}, \dots, \hat{s}_n \rangle^\omega$. The corresponding concrete loops may begin at different points and may have different lengths as is illustrated in Figure 3.

Algorithm 6 PolyRefine

```
for  $j := 1$  to  $m$  do
   $\equiv'_j := \equiv_j$ 
  for every  $a, b \in E_j$  do
    if  $proj(S_{i,0}, j, a) \neq proj(S_{i,1}, j, b)$  then
       $\equiv'_j := \equiv'_j - \{a, b\}$ 
    end if
  end for
end for
```

Thus now given a model M and an **ACTL*** specification ϕ whose counterexample is either a path or a loop, we can find an initial abstraction, then model check the abstract system. Then if the ϕ is satisfied in the abstract system, it will hold in the concrete system as well. Otherwise we check to see if the abstract counterexample produced, corresponds to a concrete counterexample. If not then we refine the abstraction and keep on doing this, till the abstract system satisfies ϕ or the counterexample generated corresponds to a concrete one.

9 Predicate Abstraction

We have seen in the previous two sections, using abstraction functions to reduce size of finite-state systems in verification. Here we explore predicate abstraction, which can be used for systems which are not finite state as well. Predicate abstraction combines the power of theorem proving and model checking. The abstraction is conservative. We construct an abstract system based on a set of predicates. We model check this abstract system, which will be finite.

9.1 Abstraction Method

Let C denote the set of concrete states. Let the predicate $I_C(x)$ denote the initial states of the concrete system, and $R_C(x, y)$ denote the transition relation on the concrete system, if there is a transition from x to y .

Let ϕ_1, \dots, ϕ_n be n predicates. Then the abstract domain is represented by bit-vectors of length n . If $\mathbb{N}_n = \{x \in \mathbb{N} \mid 0 < x \leq n\}$ then the bit-vectors may be defined as $\mathbb{N}_n \rightarrow \{0, 1\}$.

We now define the abstraction and concretization function.

$$\alpha : C \rightarrow (\mathbb{N}_n \rightarrow \{0, 1\}), \alpha(x)(i) = \phi_i(x)$$

$$\gamma : (\mathbb{N}_n \rightarrow \{0, 1\}) \rightarrow \text{Pred}(C), \gamma(s)(x) = \bigwedge_{i \in \mathbb{N}_n} \phi_i(x) \equiv s(i)$$

Given predicates, Q_C and Q_A over the concrete and abstract states, we can define the abstraction and concretization functions for these.

$$\alpha(Q_C)(s) = \exists x. Q_C(x) \wedge \bigwedge_{i \in \mathbb{N}_n} \phi_i(x) \equiv s(i)$$

$$\gamma(Q_A)(s) = \exists s. Q_A(s) \wedge \bigwedge_{i \in \mathbb{N}_n} \phi_i(x) \equiv s(i)$$

We can show easily from the above, that

$$\begin{aligned} X &\rightarrow \gamma\alpha(X) \\ \exists x. \gamma(S)(x) &\rightarrow (S = \alpha(\gamma(S))) \end{aligned}$$

This shows that the abstraction scheme is a Galois connection.

We can define the abstract initial states by $I_A = \alpha(I_C)$, and the abstract transition system as

$$R_A(s, t) = \exists x \exists y (\gamma(s)(x) \wedge \gamma(t)(y) \wedge R_C(x, y)).$$

If we define reachability as a functional

$S_A(t) = I_A(t) \vee S_A(t) \vee (\exists s(S_A(s) \wedge R_A(s, t)))$.

Thus the set of abstract reachable states is a fixed point of this functional. Also we can see that $S_C(x) \rightarrow \gamma(S_A)(x)$ where $S_C(x)$ denotes the predicate which is true if a concrete state x is reachable.

Computing the abstract transition relation may be inefficient. Also a large number of abstract states may not be reachable, so it is unnecessary to compute transition relations for those. So we try to compute approximate transition relation $R(x, y)$, which is an over-approximation of the actual abstract transition relation R_A , i.e. $R_A(s, t) \rightarrow R(s, t)$.

9.2 Counterexamples and Refinement

We have seen how to construct an abstract system using predicate abstraction. The choice of predicates would usually given by the user. If there is a counterexample generated using the abstract transition relation $R_A(s, t)$ defined above, we would call it a valid abstract counterexample. In the verification process we would begin with an over-approximation of the abstract transition relation and then generate successive approximate transition relations. If a counterexample generated by the approximate transition relation, does not correspond to a valid abstract counterexample, then the approximation would be refined, and this would continue, until the property to be verified, is proved or a valid abstract counterexample is generated.

Let R denote a translation relation which is an over approximation of the exact abstract transition relation. Now if this transition relation is able to prove the property then the proof is complete. Otherwise the model checker generates an abstract counterexample trace which violates the verification condition. Let this trace be denoted by s_0, s_1, \dots, s_n , such that $I_A(s_0)$ holds and find the least i for which $R_A(s_i, s_{i+1})$ does not hold. The procedure shown below shows how to get the next approximation. The algorithm tries to find a constraint $C(s, t)$, such that $R_A(s, t) \rightarrow C(s, t)$ and $C(s_i, s_{i+1})$ is false. Then the abstract transition relation $R'(s, t) = R(s, t) \wedge C(s, t)$ is also conservative and is taken as the next approximation. The routine **RefineTransRel** shown below returns such a constraint $C(s, t)$.

In [DDP99, DD01] examples of using predicate abstraction by successive approximation of ab-

Algorithm 7 RefineTransRel(s_j, s_{j+1})

```

 $X := \gamma(s_j)(x) \wedge \gamma(s_{j+1})(y)$ 
for each conjunct  $p$  in  $X$  do
  remove  $p$  from  $X$ 
  if satisfiable( $X \wedge R_C(x, y)$ ) then
    add  $p$  back to  $X$ 
  end if
end for
return  $\neg\alpha(X)$ 

```

stract transition relations, for verification of concurrent garbage collection and a secure contract signing protocol are discussed.

10 Conclusion

In this report basic framework for using abstractions in model checking is discussed. We can see that abstraction is a very useful technique, using which we can reduce very large finite-state

systems or infinite-state systems, to finite-systems of smaller size and then verify properties on these. In Sections 8 and 9, we have seen two approaches to using abstraction. First uses abstractions in symbolic model checking. In this we have seen methods to find initial abstraction and then identify spurious path and loop counterexamples. However there may be counterexamples which are neither paths nor loops. Such examples need to be considered. Also the problem of finding coarsest refinements is **NP**-complete. Efficient approximate algorithms for this, would improve the efficiency of the model checker.

In predicate abstraction, the problem is that of finding good candidate predicates. These are currently supplied by human inputs. Automatic methods of generating good predicates would be useful. Some work has been done in this area.

Acknowledgments

I would like to thank my guide Prof. Supratik Chakraborty for the consistent direction he has put into my work. The talks in CFDVS and interaction there were useful.

References

- [BCL⁺94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. MacMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.
- [Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretations : 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 Principle of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [CGJ⁺00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
- [CGL94a] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions of Programming Languages and Systems*, 16(5):1512–1542, September 1994.
- [CGL94b] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking. In *Proceedings of the International Summer School on Deductive Program Design*, volume 152 of *F*, Marktobendorf, Germany, August 1994. SPRINGER-VERLAG NATO ASI.
- [DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, Boston, USA, June 2001.

- [DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.
- [Ech] Federico Echenique. A short and constructive proof of tarski's fixed-point theorem. <http://citeseer.ist.psu.edu/645846.html/>.
- [Tar55] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.