PREDICATE ABSTRACTION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Satyaki Das
December 2003

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David L. Dill
(Principal Advisor)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dawson Engler

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Zohar Manna

Approved for the University Committee on Graduate Studies.

Abstract

Designing basic protocols, used in networking, security and multiprocessor systems is hard. All of these have to deal with concurrency, that is the actions of multiple agents in parallel. This makes their design error-prone since all possible interactions between the various agents in the system have to be considered.

In addition to concurrency, many of these protocols are designed to work with any number of replicated agents. For instance, protocols that set up routing tables in a network are designed to work irrespective of the number of nodes present. Similarly in cache coherence protocols there may be any number of client caches present.

Model checking, a method of enumerating and checking all states of a system, can be used to prove properties of concurrent protocols that are finite state. It can also be applied to finite instances of parameterized protocols and makes an excellent tool for finding bugs. However, in the general case, the system is not finite state, and so model checking can not be used to prove correctness.

For such systems, interactive theorem proving has been used to prove correctness. Theorem proving is extremely powerful since the entire arsenal of mathematical techniques are at one's disposal. However applying it requires a skilled user. For large examples, it can also be tedious.

Here new predicate abstraction techniques are described that allow for easy, and in many cases automatic, verification of safety properties of infinite state systems. A finite set of abstraction predicates defined on the concrete system are used to create a conservative finite state abstraction. The reachable state set of the conservative abstraction, is by definition, a superset of the reachable state set of the original system. The abstract version of the verification condition, a safety property, is model

checked on the abstract system. If successful the verification condition holds in the original system. Otherwise more analysis is carried out to figure out if there is a real bug in the system or if the abstraction needs to made more precise by adding extra predicates.

Acknowledgments

First I thank my adviser, Professor David Dill, for his unwavering support throughout my stay at Stanford. I have learned many neat technical things from him in addition to important lessons in facing the ups and downs of graduate student life with equanimity. Without his help this thesis would not have been possible.

I also thank my thesis committee members, Professors Dawson Engler and Zohar Manna for their help in writing my thesis better.

The research into predicate abstraction was sponsored by grants from the National Science Foundation (grant number 0121403), DARPA (contract 00-C-8015) and NASA (contract NASI-98139).

I thank my research group colleagues, Aaron Stump, Alan Hu, Chris Wilson, Clark Barrett, David Park, Husam Abu-Haimed, Jeffrey Su, Jens Skakkebaek, Madanlal Musuvathi, Robert Jones, Kannas Shimizu, Sergey Berezin, Shankar Govindraju, Ulirich Stern, Vijay Ganesh, Seungjoon Park, Norris Ip and Han Yang. They have helped me a lot by writing software that I used in my thesis work and by enduring many of my practice talks and paper drafts. I also thank the Computer Systems Lab support staff, Charles Orgish, Joe Little, Thoi Nguyen, Helen Nichols, Kersten Barney and Mina Madrigal-Torres.

Finally I thank my family; my father, Dipak, my mother, Shipra, and my little sister, Swaha for their love and support. I would never have come this far without them.

Contents

List of Tables

List of Figures

Predicate Abstraction

Satyaki Das

Chapter 1

Introduction

1.1 Infinite State Concurrent Systems

This dissertation is concerned with techniques for automatically proving the correctness of infinite state concurrent systems. *Concurrent systems* are systems that consist of multiple communicating agents executing in parallel. *Infinite state concurrent systems* are concurrent systems that have an unbounded number of states. This could arise if the concurrent system is parameterized, that is, it consists of any number of identical replicated sub-systems, or if unbounded data types, like integers, unbounded stacks and queues, are used.

At first glance, it may seem that infinite state systems are not of practical significance, since all systems encountered in real life are finite state. But many systems are designed first as parameterized systems and then instances of fixed size are realized for use. Examples of such parameterized systems include multiprocessor systems which consist of multiple identical processors executing in parallel, network protocols where the network consists of multiple identical nodes that are connected to each other and security protocols where multiple agents communicate amongst themselves. Such systems are families of systems, where the individual members of the family differ from each other only in the number of times a component has been replicated.

1.2 Verifying Infinite State Concurrent Systems

To prove properties of a concurrent system, all possible interactions among the agents have to be considered. Failure to do this systematically leads to bugs in the final system.

1.2.1 Model Checking

Model Checking is a verification technique for a finite state system, where all the states of the system are exhaustively enumerated and the correctness condition checked at each state. A big advantage of model checking is that it can be carried out automatically. This method has been successfully applied to proving the correctness of hardware designs.

Exploration of the state space can be done *explicitly*, that is all the reachable states are computed starting from the initial states, or *symbolically*, that is a logic formula denoting the sets of reachable states is computed. *Binary Decision Diagrams (BDD)* [10], which are a method of representing boolean logic formulas canonically, are used to implement symbolic model checkers.

Finite instances of infinite state systems can be exhaustively analyzed by model checking. For infinite state systems that are parameterized, a small value may be chosen for the parameter. Also for infinite state systems that contain infinite data types, model checking can be applied after imposing a limit on the size of the state variables. For instance, instead of real integers, 32 bit integers may be used and instead of unbounded stacks, a finite stack may be used. Model checking is useful for finding bugs in the system. Any problems found in the finite instance of the system will be present in the original system.

But model checking alone cannot prove that the infinite state system is correct. A lack of errors in the scaled down version does not mean that no errors are present in the original infinite state system.

1.2.2 Interactive Theorem Proving

Interactive theorem proving is a verification technique where the correctness condition of the system is written as a theorem and then proved interactively with varying degrees of manual intervention. Theorem proving is powerful, since all the methods of logic and mathematics are available.

Traditionally theorem proving has been used to proved infinite state systems correct. However theorem proving tools usually need manual intervention. This limits the complexity of the problems that can be tackled.

1.3 Predicate Abstraction

The goal of the thesis is to ease the verification of infinite state systems. Predicate abstraction combines theorem proving and model checking to achieve this.

1.3.1 Safety Properties

A *safety property* asserts that a system does not exhibit bad behavior. An example of a safety property is that the system never goes into an error state. Being able to prove safety properties is desirable, but it is just one facet of system correctness. It is also important to show that the system does something useful, and such properties are called *liveness properties*.

The methods discussed here are only applicable to proving safety properties.

1.3.2 Predicate Abstraction for Safety Properties

In infinite state systems, safety properties can be proved by induction [28]. First an *inductive invariant* has to be proved on the system. This means that the invariant holds in the initial state of the system and every possible transition preserves it, that is if the invariant holds in some state then it continues to hold in every successor state as well. Now if the inductive invariant implies the desired property then the proof is complete. Finding the inductive invariant is often the hardest part of the proof.

Consider the finite state system in which there is a boolean state variable for each of the atomic predicates present in the inductive invariant. Given a state in the original system, the corresponding state in the finite state system can be computed by evaluating each of the predicates and then constructing a bit vector out of the results. A pair of states in the finite state system are in the abstract transition relation if actual states corresponding to them are in the transition relation of the original system.

By definition, the inductive invariant must be a superset of the reachable states in the abstract system. If that is not the case then a concrete state satisfying the inductive invariant would have a successor that did not satisfy the inductive invariant, which leads to a contradiction! This shows that the set of abstract reachable states is nothing but a inductive invariant for the system. In fact it is the strongest possible inductive invariant that can be constructed by combining the constituent atomic predicates with arbitrary boolean connectives.

The predicate abstraction algorithm described here automatically constructs the strongest possible inductive invariant described in the preceding paragraph. In addition to this, a heuristic predicate discovery algorithm has been developed that is automatically able to find all the needed predicates for many of the test cases that were tried.

1.4 Overview of the thesis

In Chapter 2, predicate abstraction is introduced and a brief overview of related work is presented. Then in Chapter 3, efficient implicit predicate abstraction, which is a method where the abstract reachable states are computed directly without constructing the abstact state machine, is described. Chapter 4 describes a successive approximation scheme where an approximate initial abstract system is refined as needed. Chapter 5 describes an algorithm that heuristically determines predicates to use. Then Chapter 6 shows how the methods described in Chapter 4 can be applied to verifying operating system device drivers.

Chapter 2

Introduction to Predicate Abstraction

In this chapter, some examples that illustrate how predicate abstraction works are described. Then some basic terms used throughout the thesis are defined. Finally an example system, which is used to work out the algorithms presented in this thesis, is described. First the syntax used in specifying asynchronous systems is described.

2.1 System Description Syntax

In the predicate abstraction prototype, the concrete system is described as a list of guarded commands. Each *guarded command* consists of a boolean expression called the guard and an action statement that modifies the system state. A guarded command is said to be enabled if the guard expression evaluates to *true*. Given the current state of the system, one of the enabled guarded commands is chosen non-deterministically and the action statement is executed to compute the next state of the system.

An example guarded command is expressed as follows:

```
i<10 --> i := i+1
```

In the above guarded command, the expression `i<10` is the guard while `i := i+1` is the action statement. If the guard is *true* then incrementing `i` would yield a possible next state of the system.

Parameterized systems are expressed as *rulesets*. A ruleset is a set of rules that have one or more parameters. The rule is replicated once for each combination of parameter values. So, in effect, the parameterized rule represents more than one rule, and in some cases could represent an unbounded number of rules.

2.2 Predicate Abstraction Examples

An example that demonstrates the efficacy of predicate abstraction is described here. Suppose that there is a counter where the counter value, a is initially 0. The counter value changes according to the guarded commands in Figure 2.2. In effect the counter is incremented by one or two depending on its current value. Assume that we need to prove that the counter value never takes on the value 151.

An obvious approach to proving this would be enumerate all possible values that can be stored in the counter. In this simple case that is possible. If we are to work through the system we can easily convince ourselves that the possible values that a takes on must be in the set:

$$\{i \in \mathbf{N} | i < 10\} \cup \{j \in \mathbf{N} \mid 10 \leq j \leq 100 \wedge j \equiv 0 \pmod 2\}$$

Since the above set has 56 elements, it would take the same number of steps to compute the set of all possible values of a. Then we would be able to prove that indeed a never takes on the value 151.

With predicate abstraction we can do better. As shown in Figure 2.2, we use three predicates to create the abstract system. The predicates are just the the possible ranges of a.

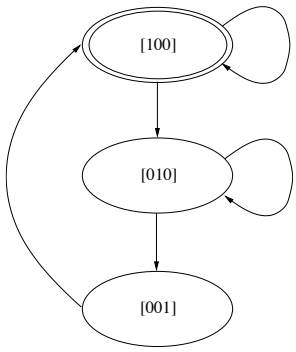
The abstract state is written as a bit-vectors of length three with the three bits representing the truth values of ϕ_1, ϕ_2 and ϕ_3 respectively. Then it is easy to see that

Initially
a = 0

Rules
$0 \leq$ a < 10 $\longrightarrow$ a := a+1;
$10 \leq$ a < 100 $\longrightarrow$ a := a+2;
$100 \leq$ a $\longrightarrow$ a := 0;

Concrete System

$\phi_1 \equiv 0 \leq a < 10$
$\phi_2 \equiv 10 \leq a < 100$
$\phi_3 \equiv 100 \leq a < 102$



(a) Abstraction Predicates (b) Abstract System

Figure 2.1: Simple Counter

the system starts in the abstract state given by the bit vector, [100] that is the state where ϕ_1 is *true* while the other predicates are *false*. From this state the counter value can only be incremented by one, so the next value must lie in the same range or it can become 10, in which case the new abstract state would be given by [010]. With similar reasoning, the rest of the transitions in the abstract system can be deduced. Now by analyzing the three state abstract system, it may be concluded that a can only take on the values such that $0 \leq a < 102$ always holds. From this we can deduce that the property of interest holds.

First of all, this example shows that the use of predicate abstraction makes at least some problems easier. This is so since we could prove the property of interest by analyzing a three-state system as opposed to a system with 56 states. In fact if the predicate $0 \leq a < 102$ was used then the verification condition could have been proved with an abstract system containing only one state! This shows that choosing

the proper set of predicates is important if predicate abstraction is going to be used on practical systems.

Another observation is that the abstract system is only an approximation of the concrete system. Just by analyzing the abstract system, it appears that the counter variable, a, can take on any integer value in the range 0 to 101, with the end points included. Clearly that is not true since a can never take on the value of any odd number in the range 10 to 100. So the simplification of the proof comes at the cost of imprecision in the abstract system. With unsuitable predicates, true properties might not hold in the abstract system produced.

This example is not really infinite state since a can only take a finite number of values. The system can with small modifications be made into an infinite state system. The system shown in Figure 2.2 is one such. It can easily be deduced that the set of possible values that a can take in the modified system is given by,

$$\{i \in N | 0 \leq i \leq 10\} \cup \{j \in N | j > 10 \wedge j \equiv 0 \pmod 2\}$$

Clearly the set of possible values of a is infinite since it can take on any even value greater than 10. As before we want to prove that a can not take on the value 151. Now a different set of predicates would be needed as shown in Figure 2.2.

Now the reachable states in the abstract system are computed. As can be seen in Figure 2.2, at least one of the predicates, $a \equiv 2 \pmod 2$ and $0 \leq a \leq 10$ is always *true*. So the verification condition holds.

2.3 Road map

Now, the work presented here can be put be in perspective. In Chapter 3 the simplest method of solving predicate abstraction is presented. In the first step, the set of all abstract initial states is computed. Now the set of abstract states reachable in one step from the initial states are computed. This process is repeated till a fixed point is reached and all the abstract reachable states have been computed. Now if the

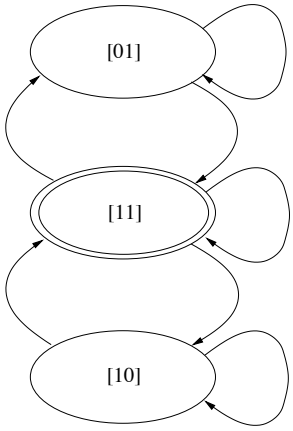
Initially
a = 0

Rules

$0 \leq$ a $<$ 10	$\longrightarrow$ a := a+1;
true	$\longrightarrow$ a := 2*a;
a $\geq$ 2	$\longrightarrow$ a := a-2;

Concrete System

$\phi_1 \equiv a(mod\ \ 2) = 0$
$\phi_2 \equiv 0 \leq a \leq 10$



(a) Abstraction Predicates (b) Abstract System

Figure 2.2: More Complex Counter

desired verification condition is a superset of the reachable set, then the verification condition holds. The technique to compute the abstract reachable states was more accurate than existing work [21]. This tool is represented by the block diagram in Figure 2.3.

This method is often less efficient that it could be because the accuracy to which the abstract reachable states are computed is not needed for the proof to succeed. Often an over-approximation of the abstract reachable states would have been sufficient to prove the verification condition. The extra accuracy requires many more calls to the validity checker than needed. It is particularly bad if some of the predicates are unnecessary. In that case the method is impractical. This brittleness is bad since automatic predicate generation techniques, described in the later chapters, can

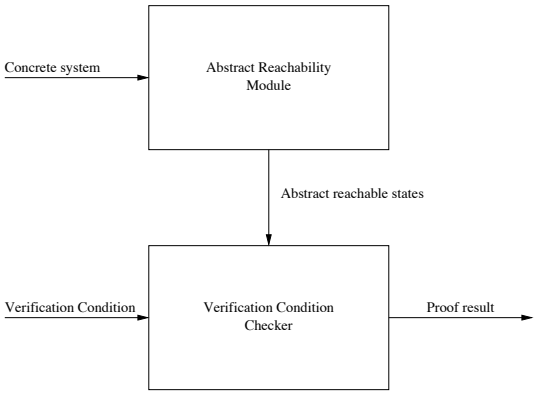


Figure 2.3: Implicit Predicate Abstraction

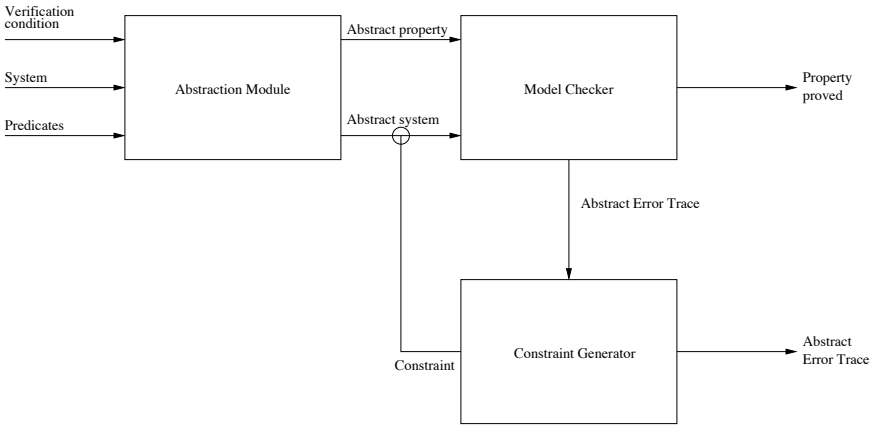


Figure 2.4: Approximate Predicate Abstraction

generate extraneous predicates.

So we developed the idea of successively refining an approximation of the abstract transition relation till it is strong enough to prove the property of interest. The advantage this has over the previous method is that it is more tolerant of irrelevant predicates. This method is described in Chapter 4. The revised block diagram for our prototype tool looks is shown in Figure 2.4.

Also note that with this approach multiple model checking runs are required. The model checking itself is very fast and it is the time taken to do the validity checks that dominate the run time of the tool. So being able to reduce the number of validity checks performed helps in making the tool more efficient.

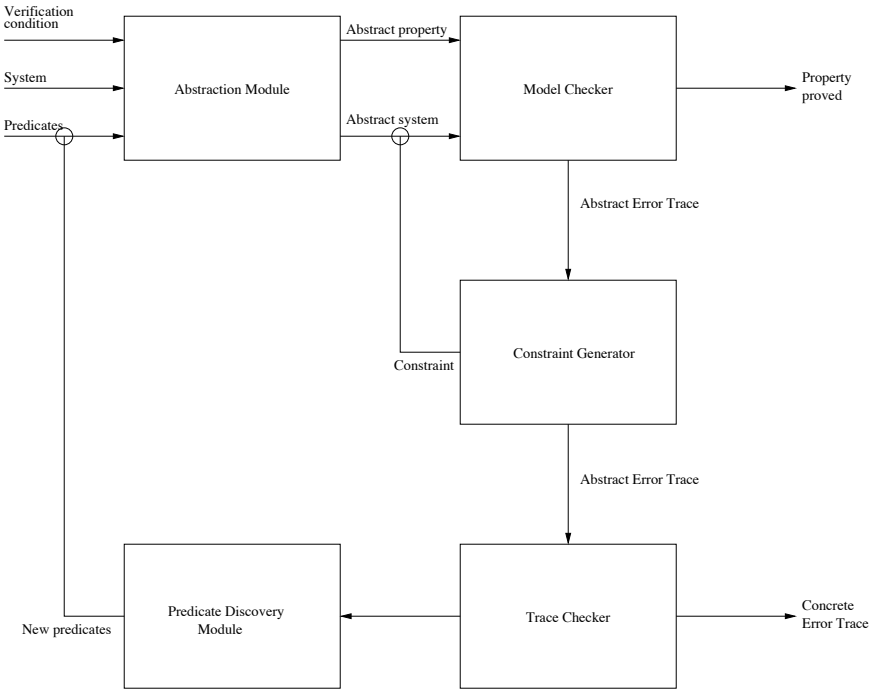


Figure 2.5: Predicate Discovery

Using the refinement method, the tool when given arbitrary predicates is able to use them to come up with an accurate enough abstract transition relation that either proves the property or produces an abstract counter-example trace that is guaranteed to be present even if the best possible abstract transition relation is used. The method is also able to gracefully deal with unrelated predicates.

Now a method was developed that can automatically find predicates to eliminate spurious abstract counter-example traces. Essentially another feedback loop is added as shown in Figure 2.5. This predicate discovery method is described in Chapter 5.

2.4 Predicate Abstraction Definitions

In what follows, logic predicates shall be used to represent sets. If the predicate, P, represents a set, then an element, x, is a member of the set if and only if $P(x)$ is *true*. As an example, the set of all even numbers is represented by the predicate, $\lambda x.\ x \equiv 0$

(mod 2). This makes it easier to formalize the proofs.

First the *abstraction* and *concretization* functions are defined. Then the abstract transition relation and the abstract initial states are defined. This completely describes the abstract system.

2.4.1 Concrete System

The set of concrete states is denoted by C. The concrete transition relation is described by an initial state predicate $I_C : C \rightarrow \mathbf{bool}$ and a concrete transition relation predicate, $R_C : C \times C \rightarrow \mathbf{bool}$. A state x is an initial state iff, $I_C(x)$ is *true*, and the state y is a concrete successor of x iff $R_C(x, y)$ is *true*.

2.4.2 Abstraction and Concretization

Let $\phi_1, \phi_2, \ldots \phi_N$ be the abstraction predicates defined on the concrete system. Then the abstract states are bit vectors of length N. The abstract set of states, A can be defined as:

$$A = \{x \in N \mid 1 \leq x \leq N\} \rightarrow \mathbf{bool}$$

Here bit vectors have been represented as a function which maps each of the indices to the corresponding bit.

Intuitively, the abstraction function maps each concrete state to an abstract state, which is a bit vector representing the values of the predicates in that state. Concretizations is the inverse of abstraction. It maps each abstract state to the set of all concrete states it represents.

Definition 1. *The abstraction and concretization functions,* $\alpha : C \rightarrow A$ *and* $\gamma : A \rightarrow C$ *are defined as,*

$$\alpha(x)(i) = \phi_i(x)$$
$$\gamma(s)(x) = \bigwedge_{i \in [1,N]} \phi_i(x) \equiv s(i)$$

($\equiv$ is the biconditional)

The definition of α and γ can be extended to work on the predicates defined over the concrete states and abstract states respectively. These extended definitions are as follows:

Definition 2. *Given predicates, Q_C and Q_A over concrete and abstract states respectively, the abstraction and concretization functions are extended as follows:*

$$\begin{aligned} \alpha(Q_C)(s) &= \exists x.\ Q_C(x) \wedge \bigwedge_{i \in [1,N]} \phi_i(x) \equiv s(i) \\ \gamma(Q_A)(x) &= \exists s.\ Q_A(s) \wedge \bigwedge_{i \in [1,N]} \phi_i(x) \equiv s(i) \end{aligned}$$

2.4.3 Abstract System

As described previously the set of states of the abstract system is A. The abstract initial states and the abstract transition relation completely define the the abstract system.

Definition 3. *The abstract initial states, $I_A : A \rightarrow$ **bool** is defined to be $\alpha(I_C)$.*

Notice that α has been used on a concrete predicate and so the second definition of α is to be used. It may be shown that the concrete and abstract initial states satisfy the inclusion relation, $I_C \Rightarrow \gamma(I_A)$

Definition 4. *The abstract transition relation is represented by a predicate $R_A : A \times A \rightarrow$ **bool** with two states, s and t as arguments. The transition relation is defined as*

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

2.5 Conservative Abstraction

The abstraction scheme described here is conservative. This means that if the verification condition holds in the abstract system then the property holds in the concrete system as well. The goal is to prove that the process of finding all reachable states is conservative. A step on the way is to prove that abstraction followed by concretization is conservative.

Lemma 1. *For any concrete state, $x : C$, it is the case that $\gamma(\alpha(x))(x)$ holds.*

Proof. Expanding the definition of γ produces,

$$\gamma(\alpha(x))(x) \quad = \quad \bigwedge_{i=0}^{N} \alpha(x)(i) \equiv \phi_i(x)$$

From the definition of α it follows that each of the conjuncts in the right hand side of the above equation is *true*. □

Now consider the abstract state, s. Assume that an arbitrary concrete state, x abstracts to it. Then it must be the case that every concrete successor of x must abstract to an abstract successor of s. This is expressed in the following lemma.

Lemma 2. *For any abstract state, s, and concrete states, x and y,*

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

Proof. To carry out the proof the left hand side of the implication is assumed to be *true*. This means that, $\gamma(s)(x) \wedge R_C(x, y)$ holds. Also applying the previous lemma to y to shows that,

$$\gamma(s)(x) \wedge \gamma(\alpha(y))(y) \wedge R_C(x, y)$$

holds. By definition of R_A it follows that, $R_A(s, \alpha(y))$ holds. Thus $\alpha(y)$ is a witness for the existential quantifier on the right hand side of the implication. □

From the first lemma above it follows that the concretization of $\alpha(I_C)$ is a superset of I_C. From the second lemma it follows that every image computation step of the model checking algorithm computes a conservative approximation of the reachable states. Hence the concretization of the abstract reachable states contains all the reachable states of the concrete system. Thus for any safety property that holds in the abstract system, its concrete counterpart holds in the concrete system.

2.6 Example

Now an example is introduced is used to illustrate how predicate abstraction works. This is a mutual exclusion protocol which was inspired by a cache coherence protocol. That protocol was simplified by removing the transactions that controlled shared access of the memory lines. The memory itself was also abstracted away leaving behind a mutual exclusion protocol based on message passing. The protocol is described in Figure 2.7 using guarded commands.

In the protocol, a $REQUEST$ message is sent by the clients contending to enter the critical section to a central server. The server receives the message only if the variable *granted* is initially *false*. The server then sets *granted* to *true* and replies to the client with a $GRANT$ message. Only one client can enter the critical section at any point of time, since the server does not receive $REQUEST$ messages if *granted* is already *true*. When the client is done, it sends a $RELEASE$ message to the server releasing access to the critical section. The server is now free to grant access to the critical section to some other client. The protocol steps are shown in Figure 2.6.

2.7 Related Work

Recently there has been a lot of work on predicate abstraction. In addition to verifying concurrent systems, predicate abstraction has also been applied to proving correctness

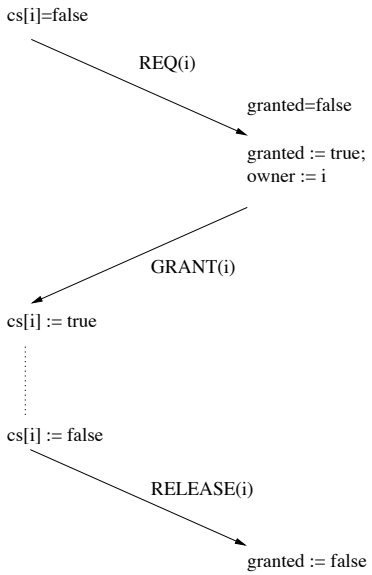


Figure 2.6: Protocol Steps

of software. Similar ideas have also been used to efficiently model check large hardware designs. The predicate discovery algorithm presented here is related to automatic invariant generation.

2.7.1 Abstraction

Abstraction is a general proof technique where a system is first simplified, then the simplified system is analyzed, and the results are transferred back to the original system. Since a simplified system is analyzed, the proof is easier to do.

Formal abstractions are used all the time in formal verification. There is actually too much work to discuss in detail. Only the work that is most closely related to predicate abstraction will be discussed here.

Abstract interpretation [16], is a form of abstraction where a potentially unbounded data type is abstracted to a finite set of points. An example of this has already been presented in Section 2.2 where a finite number of ranges have been used to model the values that a state variable of type integer can take.

Abstraction has been widely used to make model checking more efficient for large

state
N : positive_integer;
node_state : array [1..N] of enum (INIT, CS);
message_type : array [1..infinity] of enum (INVALID, REQUEST, RELEASE);
message_node : array [1..infinity] of [1..N];
owner : [1..N];
busy : boolean;

/* Client Actions */
ruleset (cell msg)
node_state[cell] = INIT ∧ message_type[msg] = INVALID →
message_type[msg] := REQUEST;
message_node[msg] := cell

node_state[cell] = INIT ∧ message_type[msg] = GRANT ∧ message_node[msg] = cell →
message_type[msg] := INVALID;
node_state[msg] := CS

node_state[msg]=CS ∧ message_type[msg]=INVALID →
message_type[msg] := RELEASE;
message_node[msg] := cell;
node_state[msg] := INIT

/* Server Actions */
ruleset (msg)
¬ busy_p ∧ message_type[msg] = REQUEST →
message_type[msg] := GRANT;
owner := message_node[msg];
busy_p := *true*

busy_p ∧ message_type[msg] = RELEASE ∧ message_node[msg]=owner →
message_type[msg] := INVALID;
busy_p := *false*

Figure 2.7: Client Actions

systems. Conservative abstraction of a finite state system [14] followed by model checking has been used to verify extremely large circuits. Abstraction has also been used to model check infinite state systems [23], though the creation of the the abstract system requires manual intervention since abstract operations corresponding to each concrete rule have to provided and proved correct.

Abstraction has been used in the context of IO Automata [27] to prove the properties of distributed algorithms [24].

2.7.2 Predicate Abstraction

Predicate abstraction, which is a special form of abstraction, was first presented by Graf and Saïdi [21]. In their work, an over-approximation of the reachable states of the abstract system was directly computed. In the work of Colón and Uribe [15], an abstract transition system was computed and then model checked. The abstract system computed was approximate, to avoid blowing up the number of validity checks needed. But this also means that there was a loss of precision; thus the method could fail to prove properties of the system that could be proved with the most accurate abstract system.

Later work in predicate abstraction [34, 33] made the predicate abstraction method incremental. As new predicates were added to an older abstraction, the new abstract system was derived from the old abstract system. Thus the abstract system was not constructed from scratch each time, thereby avoiding repeating the same proofs.

Predicate abstraction has also been extended to parameterized systems [26]. A counting abstraction was used where the number of processes that satisfied each predicate was tracked.

2.7.3 Software Model Checking

Predicate abstraction is being used in the SLAM project [7]. A abstract boolean program [6] is constructed based on the abstraction predicates. Then the boolean program is model checked [4] to check if error states are ever reachable. If the model

checking fails, a concrete error trace is generated or new predicates are produced to rule out the spurious counter-example [5].

The abstract boolean program, which is essentially a push down automata, can model recursive functions. This means that concurrent systems, that is systems with more that one thread of execution, cannot be handled. Each thread has a stack associated with it, and since a Turing machine can be simulated with two stacks, checking reachability of an arbitrary statement in such a system is equivalent to solving the Turing machine halting problem.

Predicate abstraction has also been used in the BLAST [36] tool and in the MAGIC [12] tool to verify software systems.

2.7.4 Invariant Generation

One of the techniques [28] of proving safety properties is to start with the set of error states and compute the set of all states from which an error state is reachable by computing pre-images till a fixed point is reached. If the intersection of this set with the set of initial states is empty, then no error state is reachable. For infinite state systems the fixed point computation may not terminate, so an over-approximation of the states from which an error state is reachable is computed. Our technique of predicate discovery is similar to this. Other similar methods for invariant generation [37, 8, 25, 32] have been proposed.

2.8 Tool Implementation

The predicate abstraction tool has been implemented in Common Lisp and uses CVC [1], a decision procedure for quantifier free first order logic, and BDD [10] as libraries. CVC formulas are used to represent the predicates on the concrete system while BDD expressions are used to represent predicates in the abstract system and also to implement the model checker for the abstract system.

A guarded command language, which is a subset of the Murϕ [18] language is used to describe the concrete system. Some features of the Murϕ syntax, like loops and

function calls in the action statements are not present, since constructing transition functions in an infinite system in the presence of these operations is hard.

Chapter 3

Implicit Predicate Abstraction

In *implicit predicate abstraction*, the abstract transition system is not explicitly computed. Instead the set of abstract reachable states for the abstract system is directly generated. If the concretization of the abstract reachable states implies the safety property of interest, then the verification condition holds.

3.1 Related work

Our work is based on the Graf/Saïdi abstraction scheme [21]. However, the original implementation represented the abstract state space as a set of *monomials* (a monomial is a conjunction of boolean variables and negated boolean variables). Instead, we use BDDs, which usually represent Boolean functions more efficiently. However, Graf and Saïdi also sacrificed some accuracy by representing the image of a monomial under a transition rule as a single monomial which must cover all of the states in the image of the transition rule. Our method has no such restriction. So, our verifier is more accurate, but may require more computation (which is performed more efficiently).

3.2 Predicate Abstraction Algorithm

The reachable states of the abstract system is computed by (the usual) breadth-first symbolic traversal of the reachable states. At any time, the algorithm has a logic formula (represented as a BDD) describing the current abstract reachable states. Initially, this formula is just the abstraction of the initial states of the concrete system. Then the algorithm computes the abstract successor states of these initial states using the abstract transition relation given in the previous chapter. The logical disjunction of this formula with the initial states produces the abstract states that are reachable in at most one step. This process is repeated till a fixed point is reached at which time all reachable concrete states must be present in the concretization of the abstract reachable states.

The key step of this algorithm is the image computation that finds the abstract successors of a set of abstract states. Computing the whole abstract transition relation given in the previous chapter is expensive. With N abstraction predicates, 2^{2N} satisfiability checks need to be made. The algorithm presented here tries to reduce the number of satisfiability checks to make predicate abstraction more practical.

One optimization that avoids checking if each pair of abstract states is in the abstract transition relation is to find the abstract successors of only the reachable states. Since the reachable abstract states are usually a small fraction of the total number of abstract states, this reduces the number of validity checks required.

However even with the above optimization a naive algorithm would still need an exponential number of satisfiability checks to find all the successors of a single abstract state. Most abstract states have only a few other abstract states as their successors. This fact is used to find an algorithm that performs well in practice.

First divide the set of all states into two equal halves. Then check if all the states in the first half are unreachable from the starting state. If this check succeeds then half the states have been eliminated with a single query. If the check fails, then check if none of the states in the other half are successors of the starting state. If either of these queries succeed, half of the states have been eliminated. On the other hand if neither query succeeds then further tests need to carried out. This process

$ABSTRACT\_NEXT\_STATES\ (S)$
begin
  return $H(R_C(x,y) \wedge \gamma(S)(x), 1)$
end

$H(\psi, i)$
begin
  if ($\psi(x,y)$ is unsatisfiable) **then**
    return *false*
  if $(i > N)$ **then**
    return *true*
  return $t[i] \wedge H(\psi(x,y) \wedge \phi_i(y)) \vee \neg t[i] \wedge H(\psi(x,y) \wedge \neg\phi_i(y))$
end

Figure 3.1: Abstract Next States Computation Algorithm

of subdividing sets of states into smaller and smaller sets and querying to see if none of them are successors of the current state is repeated till the exact set of successor states is computed.

This method is dependent on the manner in which the partitioning of the state space is carried out. In the tool, the partitioning is done based on the values of the abstraction predicates. In practice only a few abstract states are successors to any given abstract state. Hence this method works quite well.

The worst case for this algorithm is if every abstract state is a successor of the current abstract state. If that is so, the method needs $3 * 2^N$ theorem prover queries. This situation never arises in practice. The naive method in comparison always needs 2^N queries.

3.3 Image Computation Algorithm

The image computation used to find the abstract next states given a predicate describing the current abstract states is given by the following definition.

Definition 5. Abstract next state function *Abstract successors of the states represented by $\lambda s.S(s)$ is given by the states that satisfy the predicate:*

$$\lambda t.\exists s.S(s) \wedge R_A(s,t)$$

This is the standard way in which given the current states and the transition relation the next states are computed.

In the algorithm presented here, a binary search is used to find the abstract next states. At each step the set of candidate next states is divided into two equal parts. Then the algorithm checks if all states in the first half can be ruled out. If successful, one unsatisfiability check has eliminated half the states. Otherwise the other half is tested. The repeated sub division continues till the exact set of abstract next states have been determined. This divide-and-conquer algorithm, presented in Figure 3.1, produces the state predicate given by the following definition.

Definition 6. Operational abstract next state function *The abstract next state predicate, $\lambda t.S'(t)$ for the state predicate $\lambda s.S(s)$ is given by,*

$$S' = H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x), 1)$$

where the auxiliary function, H is defined as,

$$H(\psi, i) = \begin{cases} \lambda t.false & \text{if } \psi(x,y) \text{ is UNSAT} \\ \lambda t.true & \text{if } i > N \\ \lambda t. \quad t[i] \ \wedge H(\lambda x, y.\psi(x,y) \wedge \phi_i(y), i+1)(t) & \\ \quad \vee \ \neg t[i] \ \wedge H(\lambda x, y.\psi(x,y) \wedge \neg\phi_i(y), i+1)(t) & \text{otherwise} \end{cases}$$

Now it will be proved that the algorithm, described by Definition 6, produces the same result as the image computation, described by Definition 5.

Theorem 1. *Given the abstract states represented by S then,*

$$\lambda t.\exists s.S(s) \wedge R_A(s,t) \equiv H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x), 1)$$

where H is defined as in definition 6

Proof. Let abstract state, t_0 satisfy the lhs of the proof obligation. After existential instantiation the expression reduces to,

$$S(s_0) \wedge R_A(s_0, t_0)$$

Using the definition of R_A and existential instantiation of quantifiers therein the above reduces to,

$$S(s_0) \wedge \gamma(s_0)(x_0) \wedge \gamma(t_0)(y_0) \wedge R_C(x_0, y_0)$$

Using the definition of γ applied to concrete predicates the above simplifies to,

$$\gamma(S)(x_0) \wedge \gamma(t_0)(y_0) \wedge R_C(x_0, y_0)$$

Now $\gamma(t_0)(y_0)$ is expanded to yield,

$$\gamma(S)(x_0) \wedge R_C(x_0, y_0) \wedge \bigwedge_{i \in [1,N]} t_0[i] \equiv \phi_i(y_0)$$

This means that x_0 and y_0 are witnesses, for x and y respectively, which demonstrate

the satisfiability of the following series of formulas.

$$R_C(x,y) \wedge \gamma(S)(x)$$

$$R_C(x,y) \wedge \gamma(S)(x) \wedge (t_0[1] \equiv \phi_1(y))$$

$$R_C(x,y) \wedge \gamma(S)(x) \wedge (t_0[1] \equiv \phi_1(y)) \wedge (t_0[2] \equiv \phi_2(y))$$

$$\vdots$$

$$R_C(x,y) \wedge \gamma(S)(x_0) \wedge (t_0[1] \equiv \phi_1(y)) \wedge (t_0[2] \equiv \phi_2(y)) \ldots \wedge (t_0[N] \equiv \phi_N(y))$$

Consider the evaluation of $H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x), 1)(t_0)$ according to the definition of H. Clearly because of the satisfiability of the first formula above, the first case in the definition is not applicable. The second case is not applicable either since $N > 1$. So the third case has to be used. Now one conjunct in the disjunction is *false*, since $t_0[1]$ and $\neg t_0[1]$ can not both be *true* simultaneously. Irrespective of whether $t_0[1]$ is *true* or *false*, $H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x), 1)(t_0)$ is the same as $H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x) \wedge (t_0[1] \equiv \phi_1(y)), 2)(t_0)$. The above simplification is repeated and at each step the second argument to H is incremented by 1. This continues until it exceeds N and then the whole formula simplifies to *true*. This completes this half of the proof.

Now for the reverse direction. Assume that some abstract state t_0 satisfies the rhs of the theorem. So,

$$true = H(\lambda(x, y.R_C(x,y) \wedge \gamma(S)(x), 1))(t_0)$$

Immediately from the definition of H it follows that $\exists x, y.R_C(x,y) \wedge \gamma(S)(x)$ holds.

Also by considering all possible values of $t_0[1]$ it follows that

$$\begin{aligned} \mathit{true} &= H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x), 1)(t_0) \\ &= H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x) \wedge (t_0[1] \equiv \phi_1(y)), 2)(t_0) \end{aligned}$$

Applying this simplification repeatedly, it can be shown that,

$$\begin{aligned} \mathit{true} &= H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x), 1)(t_0) \\ &= H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x) \wedge (t_0[1] \equiv \phi_1(y)), 2)(t_0) \\ &= H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x) \wedge (t_0[1] \equiv \phi_1(y)) \wedge (t_0[2] \equiv \phi_2(y)), 3)(t_0) \\ &\vdots \\ &= H(\lambda x, y.R_C(x,y) \wedge \gamma(S)(x) \wedge \bigwedge_{i \in [1,N]} (t_0[i] \equiv \phi_i(y)), N+1)(t_0) \end{aligned}$$

Applying the definition of H to the last line of the above means that

$$\exists x, y.\gamma(S)(x) \wedge R_C(x,y) \wedge \bigwedge_{i \in [1,N]} t_0[i] \equiv \phi_i(y)$$

holds. Use the definition of γ to simplify $\gamma(S)(x)$ to get,

$$\exists s, x, y.S(s) \wedge \gamma(s)(x) \wedge R_C(x,y) \wedge \bigwedge_{i \in [1,N]} t_0[i] \equiv \phi_i(y)$$

Using the definition of γ this simplifies to,

$$\exists x, y, s.S(s) \wedge \gamma(s)(x) \wedge R_C(x,y) \wedge \gamma(t_0)(y)$$

Now using the definition of R_A conclude,

$$\exists s.S(s) \wedge R_A(s, t_0)$$

Hence t_0 satisfies the lhs of the theorem. □

3.4 Image Computation Example

To illustrate the image computation process an example image computation from the mutual exclusion protocol described before is used. The example used is the mutual exclusion protocol described in the previous chapter. To keep the example computation small, a subset of the predicates actually needed to prove correctness will be used.

Assume that there are five abstraction predicates,

1. $BUSY_A$. The cell A is busy. The predicate used is $busy[a]$.
2. $BUSY_B$. The cell B is busy. The predicate used is $busy[b]$.
3. $GRANT_0$. There are no grant messages in the queue. The predicate used is $\forall i.\neg(msg.type[i] = GRANT)$.
4. REQ_0. There are no request messages in the queue. The predicate used is $\forall i.\neg(msg.type[i] = REQ)$.
5. $GRANT_{2+}$. There are two or more distinct grant messages in the queue. The predicate used is $\exists i, j.\neg(i = j) \wedge msg.type[i] = GRANT \wedge msg.type[j] = GRANT$.

With these predicates the initial abstract state is given by the bit vector, $s_0 \equiv [00110]$. The first bit is 0 meaning that the cell A is not busy. Similarly the cell B is not busy and so the second bit is 0 as well. Also there are no messages in the queue so the last three bits are 1, 1 and 0 respectively.

The algorithm described here is used to compute the abstract next states from this initial state. The abstract next states for each guarded command is computed and then the disjunction of those states produces all the abstract successors.

The concretization of the current state yields the formula, $\psi(x)$ given by

$$\begin{aligned}
\psi(x) &= \neg x.busy[a] \\
&\wedge \neg x.busy[b] \\
&\wedge \forall i.\neg(x.msg.type[i] = GRANT) \\
&\wedge \forall i.\neg(x.msg.type[i] = REQ) \\
&\wedge \forall i,j.\neg(x.msg.type[i] = GRANT \wedge x.msg.type[j] = GRANT)
\end{aligned}$$

The algorithm first checks if any state is reachable from the initial state by testing the satisfiability of, $\psi(x) \wedge true \wedge R_C(x,y)$. The state predicate, *true* denotes the set of all states so the formula would be satisfiable if some concrete successor of a state x, which satisfies ψ exists. Since the guard of the first rule holds for every state satisfying ψ, there must be some concrete state, y, which is the next state of x. So the satisfiability check succeeds and it is time to divide the set of next states into two and check successors are present in each half. As specified in the algorithm, the first predicate is chosen to divide the set of states. Now the satisfiability of $\psi(x) \wedge y.busy[a] \wedge R_C(x,y)$ is checked. Clearly this formula is unsatisfiable. This is so since the *busy* field of none of the cells is changed if the rule is executed. This means that the no abstract state whose first bit is 1 can be among the next states of s_0. It is no surprise that $\psi(x) \wedge \neg y.busy[a] \wedge R_C(x,y)$ is satisfiable. So further investigation is needed in this case. Now the second predicate is chosen to divide the set of possible abstract successors. This process is repeated with all the remaining predicates. If the formula to be checked becomes unsatisfiable then a 0 is placed in that position of the BDD and no nodes below it are expanded. When the algorithm runs out of predicates and the formula is still satisfiable then a 1 is put in that position. The result of the computation is shown in Figure 3.4. Interpreting the resultant BDD, the abstract successors are the bit vectors, [00110] and [00100]. The first of these is nothing but

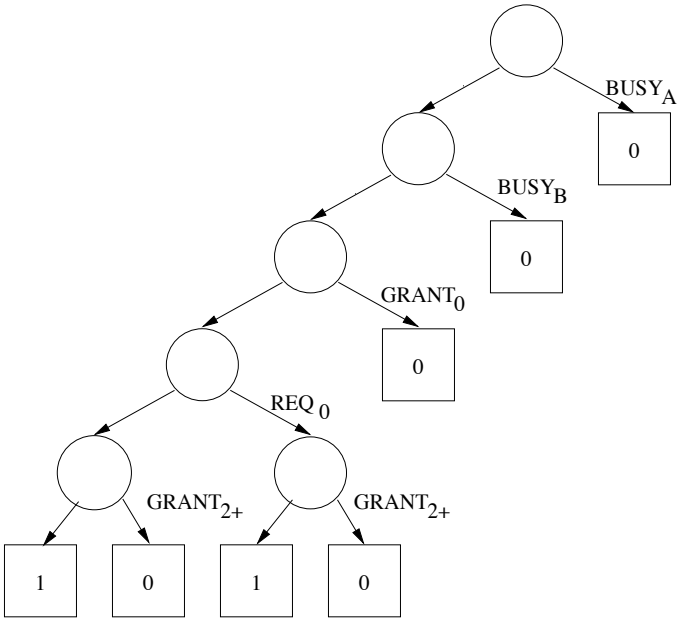


Figure 3.2: Next State Computation Example

the current abstract state. If the guard of the rule does not hold then the concrete state does not change. In this case, this can happen if the number of nodes present in the system is zero. So this returns the abstract state unchanged. The other state is more interesting. That is produced by executing the rule and differs from the initial state in that there is now a *REQ* message in the queue. So the fourth bit of the state is 0.

From the previous paragraph it is apparent that the abstract next states always contain the current abstract states. However producing the current abstract states is not necessary for correctness since the model checker has already seen them. As an optimization the implementation would check if the guard of the rule is enabled for the current concrete states. If it is not then the algorithm returns *false*, that is the empty set of states, immediately. This does not change the set of reachable states but makes the computation simpler by reducing the number of satisfiability checks.

3.5 More Realistic Examples

The predicate abstraction algorithm presented here was applied to two non-trivial problems, the FLASH Cache Coherence Protocol and the On-The-Fly Garbage Collection Algorithm

3.5.1 FLASH Cache Coherence Protocol

The model of the Stanford FLASH Cache Coherence Protocol consists of a set of nodes, each of which contains a processor, caches, and a portion of global memory of the system. Each cache line-sized block in memory is associated with a directory header which keeps information about the line. The state of a cached copy is in either *invalid*, *shared* (readable), or *exclusive* (readable and writable). The distributed nodes communicate using asynchronous messages through a point-to-point network. The objective of the protocol is to maintain consistency between the copies of data available in the caches and main memory.

This protocol has been previously verified using an aggregation abstraction with the help of a theorem prover [29]. This proof required many lemmas that showed that various pairs of actions commute (that is produce the same result irrespective of execution order). These lemmas hold in a super set of the reachable states. An inductive invariant representing this over approximation of the reachable states had to be proved. Constructing this invariant was by far the most difficult part of the proof.

Predicate abstraction was used to automatically derive this inductive invariant. The constituents of the final invariant, that is atomic properties present in the invariant, were used as the abstraction predicates. The predicate abstraction algorithm could then automatically find the inductive invariant.

One verification condition for the system is that the system has at most one exclusive copy of a memory line. To prove this the abstraction predicates needed are as follows:

- There are no exclusive copies

- There is a a single exclusive copy
- There are no PUTX replies in the network
- There is a single PUTX reply in the network

In the original protocol the directory entry maintained a count of sharers (read-only cached copies) of the corresponding memory line. When this count is zero there are no sharers. Since the implemented system can not handle counting abstractions directly, a set of nodes representing the sharers of the memory line was used.

3.5.2 On-The-Fly Garbage Collection

The On-The-Fly Garbage Collection algorithm proposed by Dijkstra, et al. [17] is widely acknowledged to be difficult to get right and difficult to prove. More detailed discussion of the history of this algorithm and subsequent variations may be found in the work of Havelund [22].

In the garbage collection algorithm, the *collector* and the user program, the *mutator* may be regarded as a concurrent system with both processes working on shared memory. The memory is abstractly modeled as a directed graph with each node having at most two outgoing edges. A subset of these nodes are called *roots* and they are special in the sense that they are always accessible to the mutator. Also any node that can be reached from one of the roots by following edges is also accessible to the mutator. The mutator is allowed to choose an arbitrary node and redirect one of its edges toward another arbitrarily chosen accessible node. Each memory node also has a *color* field which the collector uses to keep track of the accessible nodes. The collector collects *garbage* nodes (that is the nodes which are not accessible to the mutator) and adds them to the *free-list*.

The proof of correctness of the garbage collector requires several theorems about graphs. These are included as axioms to the system. They are not 'predicates' because they are true for all graphs independent of the correctness of the garbage collector algorithm.

```
begin
    while(true)
        choosesome
                n, k ∈ [0, M),
                choice ∈ boolean s.t.
                accessible(k) = true
        if choice = true ⟶
                L[n] := k;
                q := k
        □choice ≠ true ⟶
                R[n] := k;
                q := k
        fi
        if color[q] = white ⟶
                color[q] := gray fi;
    end /* while */
end
```

```
begin
    shade all roots;
    i := 0; k := M;
    do (k > 0) ⟶
       c := color[i];
       if c = gray ⟶
                    k := M;
                    shade L[i], R[i];
                    color[i] := black;
       □c ≠ gray ⟶ k := k − 1
       fi;
       i := i + 1 (mod M)
    od

    j := 0;
    do (j < M) ⟶
       c := color[j];
       if c = white ⟶ append j to free list
       □c ≠ white ⟶ color[j] := white
       fi;
       j := j + 1
    od
end
```

Figure 3.3: Mutator and Collector Algorithms

Algorithm description

The action of the mutator is to first redirect an edge of an arbitrarily selected node toward another randomly chosen accessible node. Then it colors the node gray if it was white and otherwise does nothing. This operation is called *shading*. The mutator is described in pseudo-code in Figure 3.3.

The job of the collector is to find the nodes that are not reachable from the roots and to recycle them. The algorithm starts with shading the root nodes. Then it iterates through all the nodes. If the collector finds finds a gray node then it shades its successors and colors the node black. After this the collector starts this iteration again. The collector algorithm is presented in Figure 3.3.

```
begin
    shade all roots;
    error := false;
    i := 0; k := M;
    do (k > 0) ⟶
       c := color[i];
       if c = gray ⟶
                    k := M;
                    shade L[i], R[i];
                    color[i] := black;
       □c ≠ gray ⟶ k := k − 1
       fi;
       i := (i + 1)modM
    od

    j := 0;
    do  (j < M) ⟶
       c := color[j];
       if c = white ⟶
          if accessible(j) ⟶ error := true fi
          append j to free list
       □c ≠ white ⟶ color[j] := white
       fi;
       j := j + 1
    od
end
```

Figure 3.4: Modified Collector Algorithm

The verification condition is that the collector does not free a node reachable from root. So the collector is modified to include an extra boolean state variable, *error*. It is initially *false*. If the collector frees an accessible node then it is set to *true*. So the verification condition can now be expressed as $\neg error$. The modified collector is shown in Figure 3.4.

As a first step the mutator is discarded and it is proved that the collector alone is correct. To achieve this loop invariants for the marking loop as abstraction predicates.

The predicates, rewritten in more readable syntax, are:

$$\forall x \in [0, i) : color[x] \neq gray$$

$$\forall x \in [0, M) : (color[x] = black \Rightarrow (color[L[x]] \neq white) \quad (3.1)$$

$$\forall x \in [0, M) : (color[x] = black \Rightarrow (color[R[x]] \neq white) \quad (3.2)$$

The following axiom about the function *accessible* is also needed:

$$\begin{array}{rl} & (color[0] = black) \\ \wedge & (\forall p \in [0, M) : color[p] = black \Rightarrow (color[L[p]] = black \vee color[R[p]] = black)) \\ \Rightarrow & (\forall q \in [0, M) : accessible(L, R)(q) \Rightarrow (color[q] = black)) \end{array}$$

This means that the function *accessible* [1] is the *least fixed point* of the class of functions for which if the function holds at a node then it holds at its child.

After the collector alone had been proved correct, the system is proved with a coarse-grained mutator. This mutator changes one of the edges of an arbitrary node to an accessible node and sets the color of target in one step. To prove that the system still behaved correctly more abstraction predicates were needed.

$$\forall x \in [i, M) : color[x] \neq gray$$

$$\exists y \in [0, M) : color[y] = gray$$

These predicates are needed because in the state where the collector is in the *marking phase* and the mutator can change the color of a node to gray then there must already exist a node yet to be examined by the collector which is gray. More axioms about

[1] The function accessible is actually a function of the graph structure of the nodes and so L and R are its arguments.

the *accessible* function were needed.

$$\forall p, q, r \in [0, M) : \quad (accessible(L, R)(p) \wedge accessible(write(L, q, p), R)(r)) \Rightarrow accessible(L, R)(r) \tag{3.3}$$

$$\forall p, q, r \in [0, M) : \quad (accessible(L, R)(p) \wedge accessible(L, write(R, q, p))(r)) \Rightarrow accessible(L, R)(r) \tag{3.4}$$

In the above, $write(R, q, p)$ represents an array which is the same as R except that it has the value p at index q. The axiom states that changing a pointer to point at an already accessible node does not increase the set of nodes that are accessible.

Finally the algorithm with the fine grained mutator described in Figure 3.3 is proved correct. Here the abstraction predicates defined in 3.1 and 3.2 were modified to reflect the fact that there could be edges from a black node to a white node when the mutator has redirected an edge but has not yet shaded the target. The modified predicates were:

$$\forall x \in [0, M) : (color[x] = black \Rightarrow (color[L[x]] \neq white \vee q = L[x]))$$
$$\forall x \in [0, M) : (color[x] = black \Rightarrow (color[R[x]] \neq white \vee q = R[x]))$$

In this case a more complex axiom about the function *accessible* is needed as well. The axiom states that if the root node of the graph is gray in color and all other nodes are either gray or white then, for every accessible white node, there exists a path from a gray node to it, entirely through white nodes.

$$(color[0] = gray \wedge \quad \forall x \in [0, M) : color[x] = white \wedge accessible(L, R)(x)) \Rightarrow \exists y \in [0, M) : color[y] = gray \wedge reachable\_white(L, R)(y, x) \tag{3.5}$$

This is sufficient to verify the safety property of the garbage collection algorithm.

3.6 Conclusion

In this chapter we have shown an algorithm that can be used to do accurate predicate abstraction. A more efficient method of doing predicate abstraction, as measured by the number of satisfiability checks, is presented in the next chapter.

The example with the mutual exclusion protocol introduced in Section 2.6 shows how the method can be improved. The abstract next states for the state [00110] were computed. As the reachable states of the abstract system are computed, the abstract successors of [10110] and [01110] need to be found. With the method described here, these next states have to be recomputed from scratch without reusing the work that has already been done. For instance it has already been proved that it the first bit is initially 0, then it continues to have the same value after the rule has executed. When finding the abstract successors of [01110], this fact could have been used to reduce the number of satisfiability checks.

Creating an approximate abstract transition relation that is enhanced as needed would avoid this drawback. In such a scenario, constraints can be added to the abstract transition relation that are used when the successors of other abstract states are computed.

Chapter 4

Successive Approximation of Abstraction

4.1 Introduction

In the previous chapter the abstract reachable states of the system were computed exactly by starting with the abstract initial states and computing the next states repeatedly until a fixed point was reached. This has two problems.

First, it is likely that an *over-approximation* of the reachable states, that is a superset of the actual reachable states, would suffice to prove the verification condition. Usually, but not always, it requires less work to compute an over-approximation of the abstract reachable states. For instance in the degenerate case, returning the abstract states, *true*, as the set of reachable states requires no validity checks. It is a valid, if useless, over-approximation of the abstract reachable states.

Second, similar validity checks might need to carried out at each step in the iteration. It could be that a subset of the abstract state bits never change while the remainder take on all possible values. Then at each step the algorithm would need to prove that none of the bits in the first set changed. Thus the method would essentially be doing the same satisfiability checks over and over again.

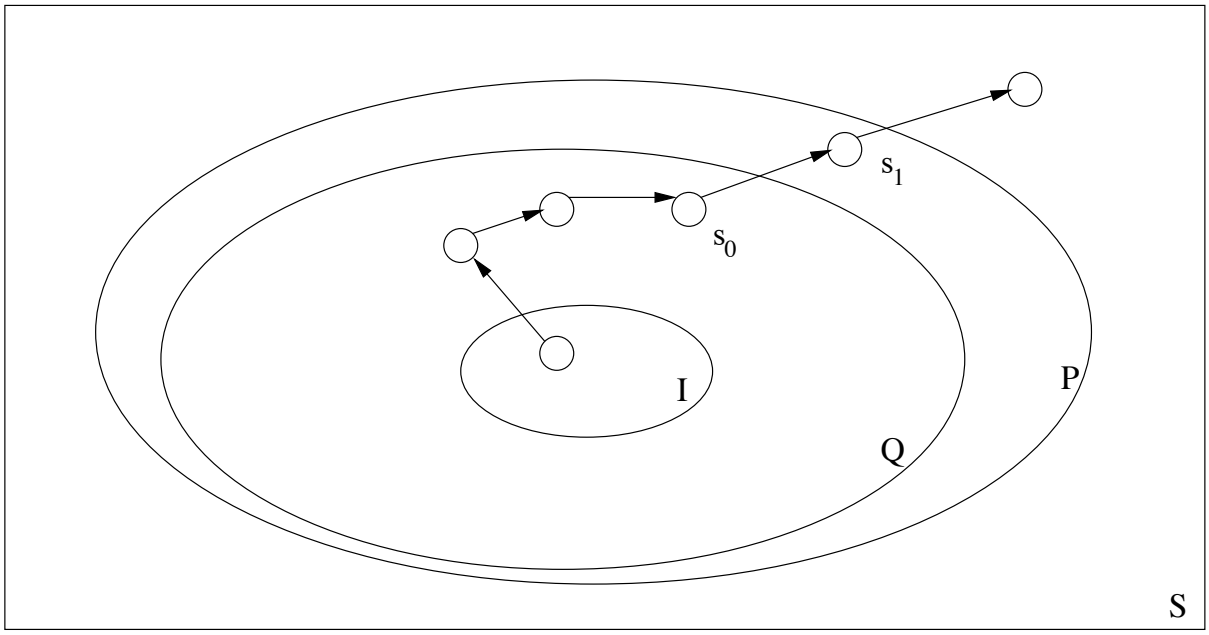


Figure 4.1: Abstract Transition Relation Refinement

In this chapter an algorithm is described that avoids both of the above problems. The algorithm starts with an over-approximation of the abstract transition relation. An *over-approximated transition relation* is a transition relation which allows all the transitions that are present in the exact transition relation. The verification condition is model checked on this abstract system. If the model checking is successful then the property is verified. Otherwise an abstract counter-example is produced. An *abstract counter-example* is a sequence of abstract states that starts in an abstract initial state, ends in an abstract state whose concretization contains a state that violates the verification condition and every pair of consecutive states are in the abstract transition relation. Now check whether each pair of successive states in the counter-example belong to the exact abstract transition relation. If they do, then the abstraction is not sufficient and an abstract counter-example that needs more analysis has been found. On the other hand, if there exists a pair of consecutive states which do not belong to the abstract transition relation, then an approximation of the transition relation has been discovered. This approximation needs to be corrected in order to prove the verification condition. So the abstract transition relation is refined

and the verification condition model checked in the refined abstract system. This process is repeated until either the verification condition is proved or an abstract counter-example is found.

4.2 Related Work

The creation of the initial abstract transition relation is similar to the abstraction method presented by Saïdi and Shankar [34]. In that work the authors construct an accurate abstract transition relation that is used in model checking. If the desired invariant does not hold, then new predicates are added. In their paper, refinement is used to construct the new abstract transition relation from the original relation. Their method computes the exact abstract transition relation which can be expensive. In contrast our strategy of successive approximation is more efficient because it attempts to compute the *least* accurate approximation that gives a definitive answer.

Colón and Uribe have also described a method that first generates an abstract transition system, then model checks it [15]. The transition relation generated is less accurate than that presented here.

The idea of counter-example-guided refinement is a generally useful technique in model checking, which has been used before, by Kurshan *et al.* [2] for checking timed automata, Balarin *et al.* [3] for language containment and Clarke *et al* [13] in the context of verification using abstraction for different variables in a version of the SMV model checker. Counter-example guided refinement has even been used with predicate abstraction by Lakhnech *et al.* [25]. However, their method refines by discovering new predicates to add, an idea that is quite different from refining the *use* of a given set of predicates in the abstract system.

4.3 Counter-Example Guided Refinement

Let us begin with an intuitive explanation of the method. Assume that the concrete state space is as shown in Figure 4.1. Here it is assumed that the verification condition,

P holds and that the abstraction predicates are sufficient to prove it. So it must be the case that all concrete the reachable states satisfy P, hence the set of states for which it holds is a superset of Q, the set of reachable states. Among the reachable states, only some are initial states, I. Since the abstract transition relation is approximate a counter-example, that would not have been present if the exact transition relation had been used, could have been produced. It starts at one of the legitimate initial states but due to the inaccuracy in the transition relation, the execution ventures out of the reachable states, Q and eventually outside the set of states that satisfy P. The first state of the trace must be inside of Q and the last state outside it. Hence the counter-example must contain at least one transition from a state s_0 inside Q to a state s_1 outside it! Clearly this transition is an artifact of the over-approximation and the abstract transition relation can be refined to eliminate it. The algorithm described here does that.

Assume that the successive approximation process starts with an over-approximation, R_0, of the exact abstract transition relation, R_A. If a state t is a successor of s in the exact transition relation then t is a successor of s in the over-approximated transition relation as well. R_0 is used to model check the verification condition. If the verification condition holds then the proof is complete. Otherwise the model checker generates an *abstract counter-example trace* which violates the verification condition. The abstract counter-example trace is a finite sequence of abstract states, $s_0, s_1, \ldots s_n$ such that $I_A(s_0)$ holds and $R_0(s_i, s_{i+1})$ holds for every $i \in [0, n)$. Also s_n violates the verification condition. Now, for each pair of consecutive abstract states, (s_i, s_{i+1}), check if $R_A(s_i, s_{i+1})$ holds. In this case, a valid abstract counter-example has been found. Otherwise R_0, can be refined to eliminate the generated counter-example. This process of model checking followed by refinement is repeated till the verification condition is proved or a valid counter-example is found.

We now explain how the refinement process works. Suppose R is the an over-approximated abstract transition relation and that the abstract counter-example trace found after model checking has two consecutive states, s_j and s_{j+1}, such that $R_A(s_j, s_{j+1})$ is *false*. The algorithm tries to find a constraint, $C(s, t)$, such that

$R_A(s,t) \Rightarrow C(s,t)$ and $C(s_j, s_{j+1})$ is *false*. Then the abstract transition relation,

$$R'(s,t) = R(s,t) \wedge C(s,t)$$

is also an over-approximate abstract transition relation. Since $R_A(s_j, s_{j+1})$ is *false*, this means that $\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y)$ is unsatisfiable for every x and every y. From the definition of γ, it follows that $\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y)$ is a conjunction of abstraction predicates, $\phi_i(x)$ and $\phi_i(y)$ and their logical complements. We wish to find a minimal subset of these predicates that is unsatisfiable when conjoined with $R_C(x,y)$. A minimal subset is desirable since it generalizes the constraint eliminating many bogus transitions similar to the one we have found. The heuristic in the present system is a simple greedy algorithm. It is explained in Figure 1.

The following theorem shows that this construction results in a new over-approximate abstract transition relation. The key point to note is that at the end of the algorithm the conjunction of the remaining conjuncts and $R_C(x,y)$ is unsatisfiable. The bit-vectors c_j and c_{j+1} determine which conjuncts have been removed. Wherever $c_j(k)$ is *false*, the conjunct involving $\phi_k(x)$ has been removed from $\gamma(s_j)(x)$ in the added constraint, $C(s,t)$. Similarly, if $c_{j+1}(k)$ is false, then the conjunct involving $\phi_k(y)$ has been removed from $\gamma(s_{j+1})(y)$.

Theorem 2. *Let the initial abstract transition relation, R satisfy $\forall s,t.\ R_A(s,t) \Rightarrow R(s,t)$ and s_j, s_{j+1} be abstract states and c_j and c_{j+1} are bit-vectors such that*

$$\begin{aligned} & \bigwedge_{i \in P} c_j(i) \Rightarrow (s_j(i) \equiv \phi_i(x)) \\ \wedge \quad & \bigwedge_{i \in P} c_{j+1}(i) \Rightarrow (s_{j+1}(i) \equiv \phi_i(y)) \wedge R_C(x,y) \end{aligned}$$

is unsatisfiable, then the new transition relation defined by,

$$\begin{aligned} R'(s,t) \;=\; & R(s,t) \wedge \\ & \neg[\bigwedge_{i\in P} c_j(i) \Rightarrow (s(i) \;\equiv\; s_j(i)) \wedge \\ & \bigwedge_{i\in P} c_{j+1}(i) \Rightarrow (t(i) \;\equiv\; s_{j+1}(i))] \end{aligned}$$

satisfies

$$\forall s,t.\;\; R_A(s,t) \Rightarrow R'(s,t)$$

Proof. To prove the theorem assume that $R_A(s,t)$ holds for some arbitrary s and t. Since $R_A(s,t) \Rightarrow R(s,t)$, it may be concluded that $R(s,t)$ holds as well. Also by definition of R_A,

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

Existential instantiation of the quantifier and using the definition of γ yields,

$$\bigwedge_{i\in P} s(i) \equiv \phi_i(x_0) \wedge \bigwedge_{i\in P} t(i) \equiv \phi_i(y_0) \wedge R_C(x_0,y_0) \tag{4.1}$$

From the precondition of the theorem it follows:

$$\begin{aligned} \neg[\exists x,y.\;\; & \bigwedge_{i\in P} c_j(i) \Rightarrow (s_j(i) \equiv \phi_i(x)) \wedge \\ & \bigwedge_{i\in P} c_{j+1}(i) \Rightarrow (s_{j+1}(i) \equiv \phi_i(y)) \wedge R_C(x,y)] \end{aligned}$$

Applying de Morgan's laws and then instantiating with x_0 and y_0 yields

$$\begin{aligned} & [\bigvee_{i \in P} c_j(i) \wedge (s_j(i) \not\equiv \phi_i(x_0))] \\ \vee \quad & [\bigvee_{i \in P} c_{j+1}(i) \wedge (s_{j+1}(i) \not\equiv \phi_i(y_0))] \\ \vee \quad & \neg R_C(x_0, y_0) \end{aligned}$$

$\neg R_C(x_0, y_0)$ is *false* from (4.1). Also, using the expressions for $\phi_i(x_0)$ and $\phi_i(y_0)$ from (4.1) yields

$$\begin{aligned} & \bigvee_{i \in P} c_j(i) \wedge (s_j(i) \not\equiv s(i)) \\ \vee \quad & \bigvee_{i \in P} c_{j+1}(i) \wedge (s_{j+1}(i) \not\equiv t(i)) \end{aligned}$$

Applying De Morgan's law and using the fact that $R(s,t)$ holds,

$$\begin{aligned} R(s,t) \quad \wedge \quad \neg[\quad & \bigwedge_{i \in P} c_j(i) \Rightarrow (s(i) \equiv s_j(i)) \wedge \\ & \bigwedge_{i \in P} c_{j+1}(i) \Rightarrow (t(i) \equiv s_{j+1}(i))] \end{aligned}$$

Using the definition of R', it follows that $R'(s,t)$ holds. □

As mentioned above, the approximate abstract system is model checked, and then refined if necessary. This process is repeated until one of the following happens:

1. The verification condition holds.

2. A counter-example trace in which for any two successive states, s_j and s_{j+1},
   $\exists x, y.\ \ \gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y)$
   holds.

It is easy to see that the process necessarily terminates in one of these situations. Every refinement removes at least one pair of abstract states from the transition relation. Since the abstract system is finite, the number of times the refinement can be carried out is bounded.

In the first scenario the desired invariant holds in an over-approximation of the exact abstract transition relation and so would also hold in the exact transition relation. Thus the desired invariant has been proved correct. In the second case the counter-example generated would also hold in the abstract machine with transition relation R_A. So further refinement of R_A would be useless. This is proved in the next theorem.

Theorem 3. *If an abstract transition system with transition relation, R such that $R_A \Rightarrow R$ and initial state set, I_A has a counter-example trace, $s_0, s_1, \ldots s_n$ such that for each $j \in [0, n)$ there are concrete states x and y (not necessarily the same for different values of j) such that,*

$$\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x, y)$$

is satisfiable, then $s_0, s_1, \ldots s_n$ is also a counter-example trace in the abstract transition system where the transition relation is R_A and the initial state set is I_A.

Proof. Since $s_0, s_1, \ldots s_n$ is an execution trace in the approximate transition system, $I_A(s_0)$ holds. Now for every $j \in [0, n)$,

$$\begin{aligned} R_A(s_j, s_{j+1}) \quad &= \exists x, y. \quad \gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \\ &\wedge \quad R_C(x, y) \end{aligned} \tag{4.2}$$

Existential instantiation of the precondition of the theorem yields,

$$\gamma(s_i)(x_0) \wedge \gamma(s_{i+1})(y_0) \wedge R_C(x_0, y_0)$$

Using this with (4.2) implies that $R_A(s_j, s_{j+1})$ is *true* and so s_{j+1} is a successor of s_j. Using this fact in conjunction with $I_A(s_0)$ proves that $s_0, s_1, \ldots s_n$ is a counter-example trace in the most precise abstract system. □

Thus, if a counter-example is generated, either the set of predicates provided are not rich enough to prove the desired verification condition or the invariant does not hold in the concrete system.

4.4 Optimizations

This section discusses optimizations that make the approach of successive refinement of approximate abstract transition relation practical.

4.4.1 Early Quantification

Early quantification is a well known technique in BDD based model checking [38, 11]. If the transition relation is expressed as a disjunction of smaller transition relations then the image computation step in BDD model checking is less likely to overflow if the image for the individual disjuncts are computed first and then the results combined than if the image computation is done for the whole transition relation.

The concrete system is described as a collection of guarded commands. The system non-deterministically chooses one of the enabled commands to generate the next state. Each of the guarded commands can be thought of as producing a transition relation. The concrete transition relation, R_C is the union[1] of these individual transition relations. So the early quantification optimization is directly applicable.

[1] in logic notation union becomes a disjunction

Manipulating the complete concrete transition relation, R_C is unwieldy since for all but the simplest systems it is too large. This leads to large formulas being produced whose satisfiability has to be checked. Instead, the tool abstracts each of the guarded commands separately to compute abstract transition relations separately for each of them. So given an abstract state, the set of abstract successors is computed by finding the next states for each of these abstract transition relations and then taking the disjunction.

The description language used to specify the concrete system allows the user to use conditional statements while writing the action. When conditionals are present the tool splits the rules into multiple rules that do not have any conditionals. This way the expressions that CVC has to check the satisfiability of are simplified.

For instance if the rule,

```
x = 2 ==>
  x := 3;
  IF (y >= 0) THEN
    y := y + 1
  ELSE
    y := 0
  ENDIF
```

is present then the tool translates this into the following pair of simpler rules:

```
(x = 2 AND y >= 0) ==>
  x := 3;
  y := y + 1
```

```
(x = 2 AND NOT (y >= 0)) ==>
  x := 3;
  y := 0
```

4.4.2 Invariant discovery

Often the predicates used to model the abstract system are interdependent and the interdependency has to be taken into account to prove the verification condition.

Such simple invariants can be provided to the model checker to limit the state space that needs to be explored.

Sometimes after the greedy minimization of the constraint, bits corresponding to just the current state might be left behind. This means that a invariant that is applicable to the whole system has been found. In this case, the model checker is informed of the constraint and the individual abstract transition relations are not modified. The advantage is that the constraint is now used globally. If an usual constraint is added then only one particular rule would be able to use this information.

4.5 Example

To illustrate how the algorithm proceeds an example is worked out here. Consider the mutual exclusion algorithm described in Section 2.6. Assume that the abstraction predicates being used are as follows:

- $state[a] = CS$
- $state[b] = CS$
- $\exists x.msg\_type[x] = GRANT$

For the purpose of this illustration, assume that the initial abstract transition relation is completely unconstrained, that is transition between any pairs of abstract states is allowed. The abstract initial state is the bit vector [000] since initially neither of the nodes a and b are in the critical section and no $GRANT$ messages are present in the message queue. The abstract version of the verification condition is that the first two bits of the abstract state are not *true* simultaneously.

Under the above conditions, the abstract system can generate the following error trace:

$$\texttt{000} \overset{2}{\Longrightarrow} \texttt{110}$$

The error trace starts in the abstract initial state and using the second client rule moves to the abstract error state. In the approximate abstract transition relation this is allowed, since all transitions are allowed in it!

It is clear that the transition should not be allowed in the abstract system. For cell a to enter the critical section there would need to be a $GRANT$ message in the message queue which is clearly not the case since the last predicate is *false*.

The transition function corresponding to the concrete rule is:

```
G(x) =  IF (x.msg_type[p] = GRANT)
        THEN
           x.msg_type[p] := EMPTY
           x.state[x.msg_node[x.msg_type[p]]] := CS
        ELSE
           x
        ENDIF
```

In the above, p is a fresh constant that models the non-deterministic choice that needs to be made because of the ruleset. Now the logic expression corresponding to $\gamma(s_1)(x) \wedge \gamma(s_2)(y) \wedge (y = G(x))$ is constructed and its satisfiability checked.

$$\begin{aligned} y = G(x) \quad \wedge \quad & (\neg x.state[a] = CS \wedge \neg x.state[b] = CS \\ & \wedge \quad \forall z.x.msg\_type[z] \neq GRANT) \\ \wedge \quad & (y.state[a] = CS \wedge y.state[b] = CS \\ & \wedge \forall w.x.msg\_type[w] \neq GRANT) \end{aligned}$$

CVC is used to check satisfiability. During the satisfiability check, CVC checks two cases. First, $x.msg\_type[p] = GRANT$ might be *false*. In this case, $y = x$ and the formula is unsatisfiable since $x.state[a] = CS$ can not be both *true* and *false* at the same time. Other wise if $x.msg\_type[p] = GRANT$ is *true* then the formula is still unsatisfiable since $(x.msg\_type[p] = GRANT) \wedge (\forall z.x.msg\_type[z] \neq GRANT)$ is a

contradiction. This means that the transition is not allowed in the exact abstract transition system.

At this point, the algorithm tries to minimize the size of the constraint to the abstract transition relation, thereby making the added constraint as strong as possible. As greedy algorithm is used to carry out this minimization.

First the formula,

$$\begin{aligned} y = G(x) \quad \wedge \quad & (\neg x.state[a] = CS \wedge \neg x.cs[b] \\ & \wedge \quad \forall z.x.msg\_type[z] \neq GRANT) \\ \wedge \quad & (y.cs[b] \\ & \wedge \quad \forall w.x.msg\_type[w] \neq GRANT) \end{aligned}$$

is checked for satisfiability. Notice that we have dropped the conjunct for the first bit in the second state to derive this formula. We are in effect checking if the state $[-10]$[2] can be a successor of [000]. Since the formula is unsatisfiable, we have been able to derive a stronger constraint. Then another conjunct is dropped and we check the satisfiability of:

$$\begin{aligned} y = G(x) \quad \wedge \quad & (\neg x.state[a] = CS \wedge \neg x.cs[b] \\ & \wedge \forall z.x.msg\_type[z] \neq GRANT) \\ \wedge \quad & (\forall w.x.msg\_type[w] \neq GRANT) \end{aligned}$$

This formula is satisfiable so we lose the contradiction. So this means that we can not add a constraint that eliminates all transitions from 000 to $--0$. However we can drop the conjunct for the last bit and the formula,

$$y = G(x) \wedge (\neg x.state[a] = CS \wedge \neg x.cs[b] \wedge \forall z.x.msg\_type[z] \neq GRANT) \wedge (y.cs[b])$$

[2]Here $-$ means unspecified. So the bit-vector denotes the pair of states 010 and 110. Hence the constraint generated is stronger.

is still unsatisfiable. So this means we can not have a transition from 000 to any state in $-1-$. We can continue dropping conjuncts from the first state too till we get the constraint that no state in $-1-$ can be a successor of -00. This means that if $CS[b]$ is initially *false* and there are no $GRANT$ messages then $CS[b]$ can not become *true* after the transition. The abstract transition relation corresponding to the concrete guarded command is given by *true* so the refined abstract transition relation becomes,

$$R'_A(s,t) = true \wedge \neg(\neg s[1] \wedge \neg s[2] \wedge t[1])$$

This process is repeated 13 times till an abstract counter-example is found that holds in the exact abstract transition relation. The verification condition cannot be proved since insufficient predicates were used.

4.6 Quantifier Instantiation Heuristics

The problems that predicate abstraction has been applied to include parameterized systems and systems that have unbounded message queues. To prove correctness of such systems, predicates containing quantifiers are needed.

For instance in the mutual exclusion example, it is an important to know how many $GRANT$ messages are present in the queue at the simultaneously. One way in which this can be expressed is to use the predicates,

$$\begin{aligned}
\phi_1 &= \exists x.(Q.type[x] = GRANT) \\
\phi_2 &= \exists x, y.(Q.type[x] = GRANT) \wedge (Q.type[y] = GRANT) \\
&\quad \wedge \neg(x = y)
\end{aligned}$$

Now the absence of GRANT messages is modeled by the predicate $\neg\phi_1$. The presence of exactly one GRANT message is indicated by the predicate $\phi_1 \wedge \neg\phi_2$ while the presence of more than one GRANT message is indicated by the predicate, ϕ_2. For this particular protocol, it is in fact the case that two GRANT messages can not be in

the message queue simultaneously. Similarly in the AODV protocol, predicates of a similar flavor, which track the presence of messages that satisfy particular conditions, are needed.

This means that the predicate abstraction tool has to be able to reason about quantifiers in order to be useful. However CVC supports a quantifier free subset of first order logic. So heuristics were needed do carry out quantifier instantiation.

4.6.1 Quantifier Instantiation Example

To demonstrate how the quantifier instantiation heuristic works, an example is described here. Suppose that the validity of the formula,

$$[\exists x.f(x) \geq 10] \vee [\forall y.f(y) \leq 15 \wedge \forall z.f(z) \leq 20]$$

The first step is to existential instantiation of the formula. For validity checks, this replaces all the universally quantified variables with fresh constants. In the above formula, y and z are replaced with y_0 and z_0 respectively.

$$[\exists x.f(x) \geq 10] \vee [f(y_0) \leq 15 \wedge f(z_0) \leq 20]$$

If the formula was initially valid then it remains valid after the transformation and vice versa. After this transformation only existential quantifiers are present.

Now, the existentially quantified sub-formulas are replaced with fresh boolean variables and the validity of the resulting formula is checked. In many cases validity of the formula does not require any quantifier instantiation. In those cases, the validity check succeeds and it is concluded that the original formula is valid too. In this example the validity of,

$$Q_0 \vee [f(y_0) \leq 15 \wedge f(z_0) \leq 20]$$

is checked. This produces the counter-example:

$$\neg Q_0 \wedge f(y_0) > 15$$

The fact that one of the boolean variables introduced in the last step appears in the counter-example means that it is possible that proper instantiation might eliminate it. If an instantiation makes the counter-example unsatisfiable then using that instantiation in the original formula would avoid that particular counter-example. So the instantiation is carried out and the check for validity repeated. Here, instantiating x with y_0 makes the counter-example unsatisfiable. So after instantiation, the formula is expanded to,

$$Q_0 \vee f(y_0) \geq 10 \vee [f(y_0) \leq 15 \wedge f(z_0) \leq 20]$$

This expanded formula is still invalid. This time the counter-example is:

$$\neg Q_0 \wedge f(y_0) < 10 \wedge f(z_0) > 20$$

This counter-example is unsatisfiable if $\exists x f(x) \geq 10$ is instantiated with z_0. Instantiating with z_0 yields the formula,

$$Q_0 \vee f(z_0) \geq 10 \vee f(y_0) \geq 10 \vee [f(y_0) \leq 15 \wedge f(z_0) \leq 20]$$

The resulting formula is valid and that implies that the original formula is valid too.

The process of finding suitable instantiations and checking validity might have to repeated many times before the formula can be proved or a counter-example is found. In general checking validity of formulas with quantifiers in them is undecidable. So, in the algorithm used, heuristics are used to find some likely candidates to try.

4.6.2 Choosing Instantiations

The quantifier instantiations that are tried by the tool are tailor made for the uses of quantified formulas that are common in predicate abstraction. So they may not be generally applicable.

In the tool, the bound variable present in the quantifiers often correspond to the ruleset parameters. The ruleset parameters are often used as array indices. So it is useful to use expressions that appear in the index position of array references.

4.7 Results

In this section the efficiency of the successive approximation based predicate abstraction is compared to that of implicit predicate abstraction. Table 4.1 shows how the number of satisfiability checks needed compares between the two methods.

For most of the small examples, the implicit predicate abstraction method does better, that is requires fewer satisfiability checks. However in the larger examples like in the Concurrent Garbage Collection Algorithm, described in Section 3.5.2 and the Contract Signing Protocol, the successive approximation method performs orders of magnitude better. This shows that this method scales better and can be applied to larger problems.

An *Contract Signing Protocol* provides a mechanism of signing contracts between two parties and guarantees some correctness conditions. The most basic of these properties is *fairness* which states that after the protocol is executed either both parties have a valid contract or neither party has a contract. Another property called *accountability* says that if any party cheats (that is deviates from the correct sequence of actions) then a trail of network messages will show unambiguously which party has cheated. Another more complex property of the protocol is *abuse freeness*. This says that while the protocol is being executed neither party should be able to obtain evidence that a contract is being signed. The protocol we looked at was introduced in [20]. It has been exhaustively analyzed for weaknesses using a model checker [35] and a fixed number of participants. A problem was discovered during this which was

Example	Number of satisfiability checks	
	Implicit	Approximate
Alternating Bit Protocol	237	153
Bakery Mutual Exclusion	27	118
Bounded Retransmission Protocol	404	333
Burns Mutual Exclusion	1065	773
Simple Cache Coherence	22	21
Mux-AST	272	203
Peterson	269	256
Semaphore	282	334
Sifakis '79	1374	718
Ticket	3405	2406
Dijkstra Mutual Exclusion	14535	3005
Concurrent GC	8000+	2275
Secure Contract Signing	266506	18513
AODV Routing Protocol	10000+	2134

Table 4.1: Different approaches to Predicate Abstraction

fixed. Here we have looked at the fixed protocol and proved that it maintains the fairness property with arbitrary numbers of participants.

The *AODV protocol* in the Table 4.1 refers to a simplified version of the Ad hoc On-demand distance Vector (AODV) routing protocol [30, 31]. The simplification was to remove timeouts from the protocol since we could not find a way of reasoning about them in our system. The protocol is used for routing in a dynamic environment where networked nodes are entering and leaving the system. The main correctness condition of the protocol is to avoid the formation of routing loops. This is hard to accomplish and bugs have been found [9]. Finite instances of the protocol has been analyzed with model checkers and a version of the protocol has been proved correct using manual theorem proving techniques.

4.8 Conclusion

In this chapter we have presented an efficient, yet precise, method of doing predicate abstraction. This method also makes predicate abstraction less susceptible to

problems in the presence of unused predicates. This means automated methods of predicate generation, that occasionally produce unneeded predicates can be used with this method. In the next chapter a method to generate interesting predicates that can be used for predicate abstraction is introduced.

```
PROVE_VERIFICATION_CONDITION(property)
begin
    I_A := Initial State predicate
    R_A := true
    while (true)
        R_orig := R_A
        trace := model check property in abstract system, (I_A, R_A)
        if empty(trace) then
          return PROPERTY_PROVED
        else
          for each pair of successive states s_j, s_{j+1} in trace do
              if γ(s_j)(x) ∧ γ(s_{j+1})(y) ∧ R_C(x, y) is unsatisfiable
              then
                R_orig := R_A
                R_A := R_A ∧ REFINE_TRANS_REL(s_j, s_{j+1})
                break
              endif
          end
          if R_A = R_orig return trace
        endif
    end
end

REFINE_TRANS_REL(s_j, s_{j+1})
/* The function returns the constraint C */
begin
    X := γ(s_j)(x) ∧ γ(s_{j+1})(y)
    for each conjunct, p in X do
        remove p from X
        if satisfiable(X ∧ R_C(x, y)) then
          add p back to X
        endif
    end
    return ¬α(X)
end
```

Figure 4.2: Abstract State Machine Refinement

Chapter 5

Predicate Discovery

The previous chapters describe a practical algorithm that, given the right predicates constructs the best possible abstraction. This chapter addresses the problem of *discovering* suitable predicates.

5.1 Introduction

In much of the work on predicate abstraction, the predicates were assumed to be given by the user, or they were extracted syntactically from the system description (for example, predicates that appear in conditionals are often useful). It is obviously difficult for the user to find the right set of predicates It is a trial-and-error process involving inspecting failed proofs. The predicates appearing in the system description are rarely sufficient. There has been less work, and less progress, on solving the problem of finding the right set of predicates. In addition to the challenge of finding a sufficient set of predicates, there is the challenge of avoiding irrelevant predicates, since the cost of checking the abstract system usually increases exponentially with the number of predicates.

In our system quantified predicates are used to deal with parameterized systems. In a parameterized system, it is often necessary to find properties that hold for all values of the parameter. For instance, if a message queue is modeled as an array

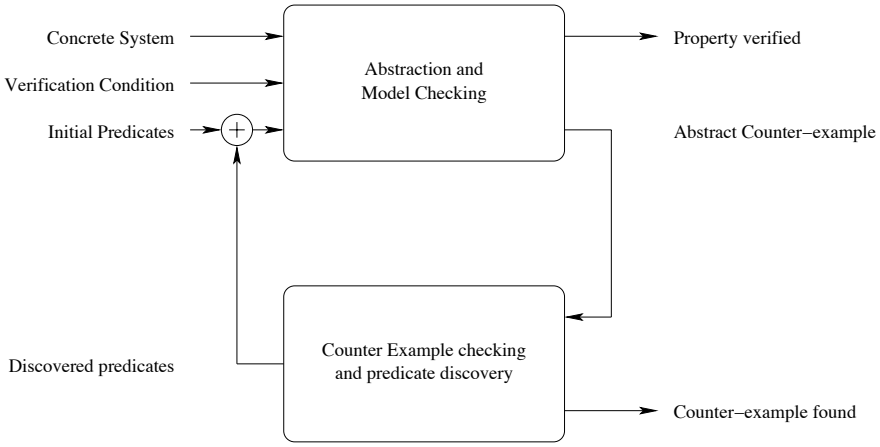


Figure 5.1: Predicate Abstraction Algorithm

and rules parameterized by the array index are used to deliver messages, then the absence of certain kinds of messages is expressed by a universally quantified formula. So predicates containing quantifiers are used.

This chapter describes new ways of automatically discovering useful predicates by diagnosing failed proofs. The method is designed to find hard predicates that do not appear syntactically in the system description, including quantified predicates, which are necessary for proving most interesting properties. As importantly, it tries to avoid discovering useless predicates that do not help to avoid a known erroneous result. Furthermore, the diagnosis process can determine whether a proof fails because of a genuine violation of the property by the actual system. If the algorithm terminates, it produces a proof that no error state is reached or a counter-example in the concrete system. The algorithm fails to terminate in some cases, as it must, because the reachability if an error state in the systems we model is obviously undecidable.

At a high level, our tool is implemented as as shown in Figure 5.1. The upper block is the tool described in chapter 3. Given a set of abstraction predicates, a verification condition and the concrete system description, it first computes an approximate abstract model. This abstract model is model checked and the abstract system refined appropriately if model checking finds that an error state is reachable that is in fact unreachable in the exact abstract transition relation. Notice that this refinement does not change the set of abstraction predicates and concentrates on using the existing

predicates more efficiently. Finally this process terminates with either the verification condition verified (in which case nothing else needs to be done) or with an abstract counter-example trace.

This chapter describes the lower block in the diagram, which checks the existence of a concrete counter-example trace corresponding to the abstract trace. If a concrete trace exists, then the verification condition is violated and an error is reported; otherwise, new predicates are discovered which avoid this counter-example.

The new predicates are added to the already present abstraction predicates and the process starts anew. Since only new predicates are used, the abstraction and model checking block reuses information from previous runs to avoid having to do the same work repeatedly.

5.1.1 Related Work

Counter-example guided refinement is a generally useful technique. It has been used in by Kurshan *et al.* [2] for checking timed automata, Balarin *et al.* [3] for language containment and Clarke *et al* [13] in the context of verification using abstraction for different variables in a version of the SMV model checker. Counter-example guided refinement has even been used with predicate abstraction by Lakhnech *et al.* [25]. Invariant generation techniques have also used similar ideas [37, 8]. Invariant generation techniques generally produce too many invariants, many of which are not relevant to the property being proved. This can cause problems with large systems. The counter-example guided refinement techniques do not produce the quantified predicates that our method needs.

Predicate abstraction is also being used for software verification. Device drivers are being verified by the SLAM project [7]. The SLAM project has used concrete simulation of the abstract counter-example trace to generate new predicates. The BLAST project [36] also uses spurious counter-examples to generate new predicates. Predicate abstraction has also been used in software verification as a way of finding loop invariants [19]. These systems do not deal with parameterized systems, hence they do not need quantified predicates.

5.2 Predicate Discovery Algorithm

Model checking the abstract system constructed in the previous chapter could yield a abstract trace. Such a trace is called a *real trace* if there exists a concrete trace corresponding to it. Conversely if there are no concrete traces corresponding to the abstract trace then it is called a *spurious trace*.

First the trace is minimized to get a *minimal spurious trace*. A minimal spurious trace is defined to be an abstract trace which is

1. spurious (no corresponding concrete trace exists.)
2. minimal (removing even a single state from either the beginning or end of the trace makes the remainder real.)

5.2.1 Checking the Abstract Counter-Example Trace

There is a concrete counter-example trace $x_1, x_2, \ldots x_L$ corresponding to the abstract counter-example trace, $s_1, s_2, \ldots s_L$ if these conditions are satisfied:

1. For each i, $\gamma(s_i)(x_i)$ holds. This means that each concrete state x_i corresponds to the abstract state s_i in the trace.
2. $I_C(x_1) \wedge \neg P(x_L)$ holds. The concrete counter-example trace starts from a initial state and ends in a state which violates P.
3. For each $i \in [1, L)$, $R_C(x_i, x_{i+1})$. For every i, x_{i+1} is the successor of x_i.

The conditions (1) and (3) determine that a concrete trace corresponding to the abstract trace exists and condition (2) determines that the trace starts from the set of concrete initial states and ends in a state that violates the verification condition.

To simplify the analysis, let $R_I(x, y) = I_C(y)$ be a transition relation that generates the initial concrete states. So $R_I(x, y)$ implies that y is an initial state irrespective of x.

An abstract state, $s_0 = true$ is appended to the beginning of the abstract counter-example trace. The transition relation, R_I is used to generate the state, x_1. So

the abstract counter-example trace now becomes, $s_0, s_1, \ldots s_L$. Since all the atomic predicates of P are present among the abstraction predicates the condition $\neg P(x_L)$ is implied by $\gamma(s_L)(x_L)$. Hence, if the formula

$$\bigwedge_{i=0}^{L} \gamma(s_i)(x_i) \quad \wedge \quad \bigwedge_{i=0}^{L-1} R_C(x_i, x_{i+1}) \tag{5.1}$$

is satisfiable then the abstract counter-example trace is real. Otherwise there is no satisfying assignment and the abstract counter-example trace is spurious. To simplify the presentation assume that the same transition relation, R_C can be used for each of the concrete steps including the first where R_I is actually used. In the implementation the first step is handled specially and R_I is used instead of R_C.

The test for spuriousness is completely a property of the transition relation and the trace itself and does not depend either on the initial states or the verification condition. So the definition of spuriousness is generalized to partial traces. A partial trace is spurious if Formula 5.1 is unsatisfiable.

5.2.2 Predicate Discovery

To understand predicate discovery we must first understand when predicate abstraction produces a spurious counter-example. Assume that in Figure 5.2 the whole abstract trace $s_1, s_2, \ldots s_L$ is spurious but the partial trace $s_2, s_3, \ldots s_L$ is real. So there are two kinds of concrete states in $\gamma(s_2)$:

1. Successor states of states in $\gamma(s_1)$.

2. States (like x_2 in the figure) that are part of some concrete trace corresponding to $s_2, \ldots s_L$.

It must be the case that the above two types of states are disjoint. Otherwise it would be possible to find a concrete trace corresponding to the whole trace thereby making it real. If predicates to distinguish the two kinds of states existed then the spurious counter-example would be avoided. In the method described here, the discovered predicates characterize states of the second type above.

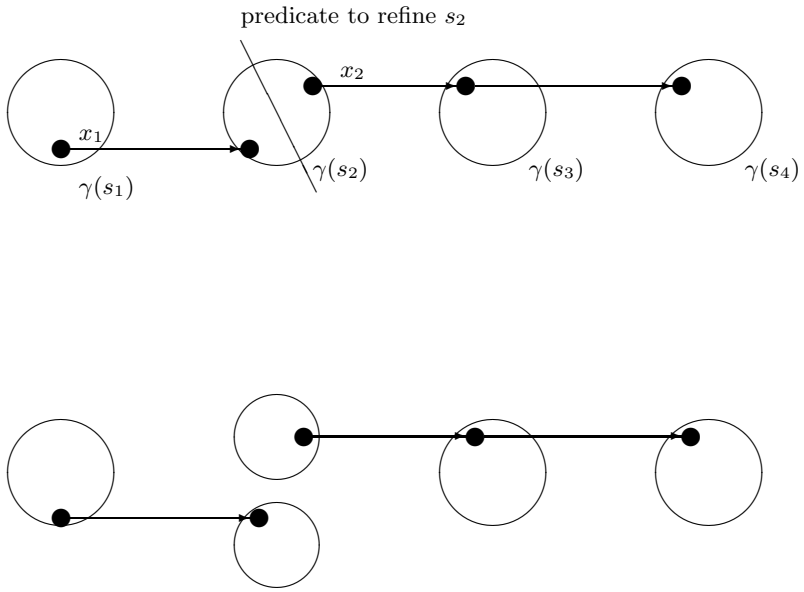


Figure 5.2: Abstraction Refinement

Once it has been determined that the abstract counter-example is spurious, states are removed from the beginning of the trace while still keeping the remainder spurious. When states can no longer be removed from the beginning, the same process is carried out by removing states from the end of the trace. This process eventually produces a minimal spurious trace.

Now consider the minimal spurious trace, $s_1, s_2, s_3, \ldots s_L$ shown in Figure 2. Here the circles representing $\gamma(s_1)$, $\gamma(s_2)$ etc. are sets of concrete states while the black dots inside the sets represent individual concrete states.

Since the trace $s_2, s_3 \ldots s_L$ is real,

$$Q_0 \;=\; \bigwedge_{i=2}^{L} \gamma(s_i)(x_i) \;\wedge\; \bigwedge_{i=2}^{L-1} R_C(x_i, x_{i+1}) \tag{5.2}$$

is satisfiable for some concrete states, $x_2, x_3, \ldots x_L$.

Our predicate discovery method depends on a property of the transition relation that has not been discussed up to this point. In the implemented tool, guarded commands are used to represent the concrete system. This means that the transition

relation is actually expressed in terms of a transition function, F_C. So the state x_3 is $F_C(x_2)$, the state x_4 is ${F_C}^2(x_2)$ [1] and so on

The goal is to find all conditions that hold in state x_2. CVC is used to do this. Using the transition function, F_C, Formula 5.2 can be rewritten as

$$Q_0 \;=\; \bigwedge_{i=2}^{L} \gamma(s_i)({F_C}^{i-2}(x_2))$$

In the above, ${F_C}^0(x_2)$ is defined to be x_2.

Now CVC is queried about the satisfiability of Q_0. This returns a finite conjunction of formulas, $\psi_1(x_2) \wedge \psi_2(x_2) \wedge \ldots \psi_K(x_2)$ which implies Q_0. So each ψ_i is a condition that any x_2 must satisfy for it to be the first state of the concrete trace corresponding to $s_2, s_3, \ldots s_L$. Now it must be the case that,

$$\gamma(s_1)(x_1) \wedge x_2 = F_C(x_1) \wedge \bigwedge_{i=1}^{K} \psi_i(x_2)$$

is unsatisfiable. Otherwise it would be possible to find a concrete trace corresponding to $s_1, s_2, \ldots s_L$! More specifically, if the predicates $\psi_1, \psi_2, \ldots \psi_K$ are added to the set of abstraction predicates, and the verifier rerun, this particular spurious abstract counter-example is not generated. This shows an automatic way of discovering a new abstraction.

However it is possible to reduce the number of additional abstraction predicates. In fact it is quite likely that not all of the predicates $\psi_1, \ldots \psi_K$ are needed to avoid the spurious counter-example. The satisfiability of the above formula is checked after leaving out the $\psi_1(x_2)$ expression. If the formula is still unsatisfiable then $\psi_1(x_2)$ is dropped altogether. The same procedure is repeated with the other ψ_i's until an minimal set of predicates remain, such that dropping any one of them makes the formula satisfiable. Notice that there may be multiple essential sets of predicates that make the above formula unsatisfiable. This method finds one such set.

[1] Here $g^i(x)$ is the result obtained by applying the function, g, i times to x.

Now consider the effect that the abstraction refinement has on the abstract system. The original abstract state, s_2 is split into two – in one part all the added predicates hold while in the other part at least one of the assertions does not hold. Also, in the abstract transition relation, the transition from the state s_1 to the first partition of s_2 is removed. It is still possible that there is a path from s_1 to s_3 through the other partition of s_2. However the refined abstraction never generates a spurious counter-example in which a concrete state corresponding to s_1 has a successor that satisfies all the assertions $\psi_1, \psi_2, \ldots \psi_K$.

5.2.3 Parameterized rules and quantified predicates

When proving properties of parameterized systems, quantified predicates are needed. These quantified predicates cannot be found either from the system description or by existing predicate discovery techniques. Invariant generation methods do find quantified invariants which may be useful in some cases. But these methods generate many invariants, and there is no good way of deciding which ones are useful.

In the presence of parameterized rules, the predicate discovery works exactly as described above. But the parameters (which are explicitly not part of the concrete state) in the rules may appear in the predicates finally generated. Recall that the predicates discovered characterize the set of states like x_2 (in Figure 2) that are part of a real abstract trace. Appearance of a rule parameter in these expressions implies that the parameter must satisfy some conditions in the concrete counterpart of the abstract trace. Any other value of the parameter which satisfies the same conditions could produce another concrete trace. Therefore, an existential quantifier wrapped around these expressions would find a predicate that is consistent with all possible behaviors of the (possibly unbounded) parameter.

Quantifier scope minimization is carried out so that smaller predicates may be found. In some cases the existential quantifiers can be eliminated all together. Often predicates of the form, $\exists x.\ Q(x) \wedge (x = a)$ where a is independent of x, are discovered. Heuristics were added so that this predicate would be simplified to $Q(a)$.

To illustrate the way quantified predicates are discovered automatically a really

state
N: positive_integer
status : array [N] of enum {$GOOD$, BAD}
error : boolean

initialize
status := All values are initialized to $GOOD$
error := *false* /* No error initially */

rule(p : subrange [1..N])
(status[p] = BAD) $\Rightarrow$ error := *true*

property
$\neg error$

Figure 5.3: Quantified Predicate Example

trivial example is presented in Figure 3. In the example system it is desirable to prove that *error* is always false. So the initial abstraction predicates are just the atomic formulas of the verification condition, in this case the predicate: $B_1 \equiv error$. With this abstraction the property cannot be proved and an abstract counter-example trace, $\neg B_1, B_1$ is returned. Since the initialization rule is handled like any other rule (only with implicit guard *true*) the abstract counter-example that is analyzed is, $true, \neg B_1, B_1$. Using the consistency test described earlier, the counter-example is shown to be a minimal spurious trace. Also the partial trace, $\neg B1, B1$ is real (that is a concrete counterpart exists) when $status[p_0] = BAD$ holds (p_0 is the specific value of the parameter chosen). However the initialization rule specifically sets all the elements of the *status* array to $GOOD$. Hence the predicate discovered is, $status[p_0] = BAD$. But notice that the parameter appears in the predicate. Hence the new predicate is, $B_2 \equiv \exists q.\ status[q] = BAD$.

The abstraction is refined with the extra predicate. The additional bit is initialized to *false*. Now the transition rule is enabled only when the new bit is *true*. Since that never happens, the rule is never enabled, and the desired property holds.

5.3 Example

The predicate discovery algorithm can be illustrated better with an example predicate discovery run from the toy mutual exclusion protocol discussed earlier. As an example, the steps that the predicate discovery algorithm goes through for the second iteration are carried out manually. At this point, there are three predicates:

- $state[a] = CS$
- $state[b] = CS$
- $\exists x.msg\_type[x] = GRANT$

The first two predicates are sub-formulas of the verification condition and so were automatically included at the beginning of the predicate abstraction process. The third predicate, which says that there is at least one $GRANT$ message in the network, was found automatically in the first iteration. Now the techniques presented in the previous two chapters are used to compute the abstract system and to model check it. Not surprisingly this produces a spurious abstract counter-example trace. The trace produced is as follows:

$$[000] \longrightarrow [001] \longrightarrow [011] \longrightarrow [111]$$

The initial state of the system is all zeros, since each of the predicates are initially false. Also there are concrete witnesses corresponding to each of the transitions. The $[000] \longrightarrow [001]$ transition means that the server had received a $REQUEST$ message and has sent its reply. The [001] state implies that there is at least one $GRANT$ message in the message queue. One of these could be addressed to B and in the second transition above B receives it and enters the critical section. Since there could have been multiple $GRANT$ messages in the queue, the third predicate could continue to remain *true*. The last step is similar to the previous one, except that this time it is A that receives a $GRANT$ message and enters the critical section.

Now the first step is to check if the abstract counter-example trace is spurious. In this case that is indeed the case. The reason for this is as follows. Initially, there

are no $GRANT$ messages in the message queue, since the third bit is 0. Then the server grants access of the critical section to some client and the message queue now has exactly one $GRANT$ message. Now when the client, B receives the $GRANT$ message the third bit should go back to 0 since the only $GRANT$ message in the message queue has been consumed. Since this has not happened in the abstract trace, it is spurious.

The next step is to find the minimal spurious trace. It is easy to see that the trace,

$$[000] \longrightarrow [001] \longrightarrow [011]$$

is the minimal spurious trace. It is minimal since removing either of the two transitions produces a trace that is not spurious, and the trace is spurious due to the reason given in the previous paragraph.

The following conditions hold in the first state of the non-spurious tail of the trace:

1. $\neg(state[a] = CS)$
2. $\neg(state[b] = CS)$
3. $msg\_type[x_0] = GRANT$
4. $msg\_type[x_1] = GRANT$
5. $msg\_node[x_1] = B$
6. $\neg(x_0 = x_1)$

The first two assertions are derived directly from the fact that the fact that the first two bits are 0. The third assertion is produced by existential instantiation of the quantifier. The fourth and fifth assertions arise from from the fact that a $GRANT$ is received by the client, B. Finally, after the $GRANT$ has been received there is at least one more $GRANT$ message. Thus the last assertion says that x_0 and x_1 are two distinct $GRANT$ messages.

The next step is to check if the above set of assertions is inconsistent with the concrete state produced by the first transition. Clearly, in this case, that is true

since in the first state there are no $GRANT$ messages in the message queue and the transition where the server grants the critical section to a client can only add one $GRANT$ message. To minimize the number of predicates the algorithm tries to remove predicates from the above list while still maintaining the inconsistency. Here the only minimal set possible is:

1. $msg\_type[x_0] = GRANT$
2. $msg\_type[x_1] = GRANT$
3. $\neg(x_0 = x_1)$

Notice that the constants, x_0 and x_1 that are not present in the concrete system description appear in the above. So in the final predicate that is added to the abstraction, quantifiers are needed. Hence the added predicate is:

$$\exists x_0, x_1.\ msg\_type[x_0] = GRANT \wedge msg\_type[x_1] \wedge \neg(x_0 = x_1)$$

This predicate that has been discovered by the algorithm is an important one. The negation of the predicate, which expresses the property that there can not be more than one $GRANT$ message in the message queue simultaneously, is an invariant for the system and it is critical in proving that mutual exclusion is maintained. This again highlights that during the predicate discovery process complete invariants need not be found, instead interesting predicates that can be combined with other predicates to produce invariants are sufficient. With these building blocks as raw material, the predicate abstraction process is able to synthesize the necessary invariants, which in a manual proof the user would have to provide.

5.4 Results

In this section the examples to which the automatic predicate discovery method has been applied are presented. In Table 5.1 below, the number of predicate discovery

Example	Predicate discovery cycles	Predicates needed
Alternating Bit Protocol	13	18
Bakery Mutual Exclusion	8	10
Bounded Retransmission Protocol	16	22
Burns Mutual Exclusion	8	12
Simple Cache Coherence	1	3
Dijkstra Mutual Exclusion	11	18
Mux-AST	6	12
Peterson	4	7
Semaphore	6	8
Sifakis 79	5	8
Ticket	13	16
AODV Routing Protocol	14	19

Table 5.1: Predicate Discovery Results

cycles is reported. All of them were completed automatically, starting with no predicates, except for the Mux-AST protocol which needed one manual predicate. This predicate was a linear invariant. It can not be discovered by our predicate discovery algorithm.

Chapter 6

Device Driver Verification

Some of the techniques of the previous chapters are used in the SLAM [7] system, which is used to verify the device drivers that are part of the operating systems written at Microsoft.

6.1 Introduction to Slam

The SLAM tool consists of several components developed over several years by researchers at Microsoft. The block diagram is shown in Figure 6.1.

First the C2BP [6] tool is used to construct a boolean abstract program from the original driver program based on the abstraction predicates. Then the model checker, BEBOP [4] is used to model check the verification condition. If the property is verified in the abstract system then by construction it holds in the original driver. Otherwise an abstract error trace is returned. The NEWTON [5] tool is used to check if there is a concrete trace corresponding to the abstract trace. If such a trace is present then a bug has been uncovered. Otherwise NEWTON generates new predicates to produce a more faithful abstraction. Then the loop is started anew.

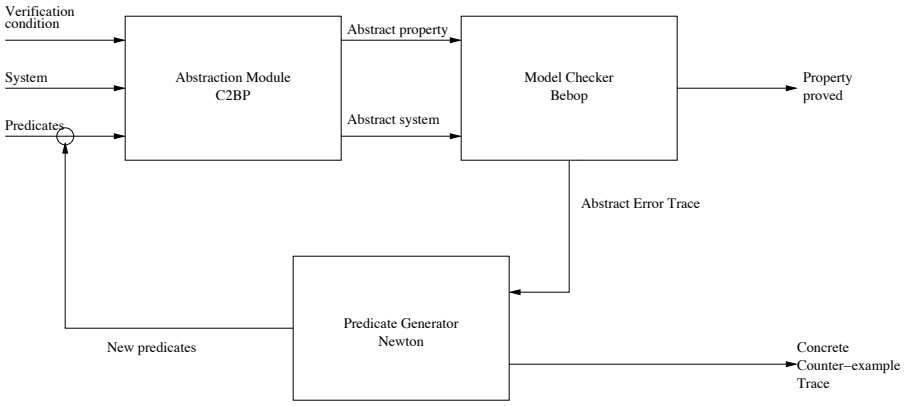


Figure 6.1: SLAM Block Diagram

6.2 Improving Slam

This chapter describes the integration of the abstract transition relation refinement method presented in Chapter 4. With this enhancement, the tool can verify larger, in terms of program size, device drivers.

The accuracy with which C2BP constructs the abstract boolean program is extremely important for optimal use of the tool. If the boolean program is too much of an over-approximation, then the model checker fails to prove the verification condition correct even if all the required predicates are present. On the other hand, constructing the exact abstract boolean program is expensive in the number of satisfiability checks that need to be carried out. In fact for many of the larger drivers, constructing the exact abstract system is not feasible. Thus there is a trade-off that needs to be made in terms of the accuracy that C2BP must achieve and the speed at which it performs.

The solution is to first construct an approximate abstract system and then refine it as needed, as discussed in Chapter 4. The revised SLAM tool-chain is described in Figure 6.2. Here we describe the results of adapting the successive refinement of the abstract transition relation to SLAM.

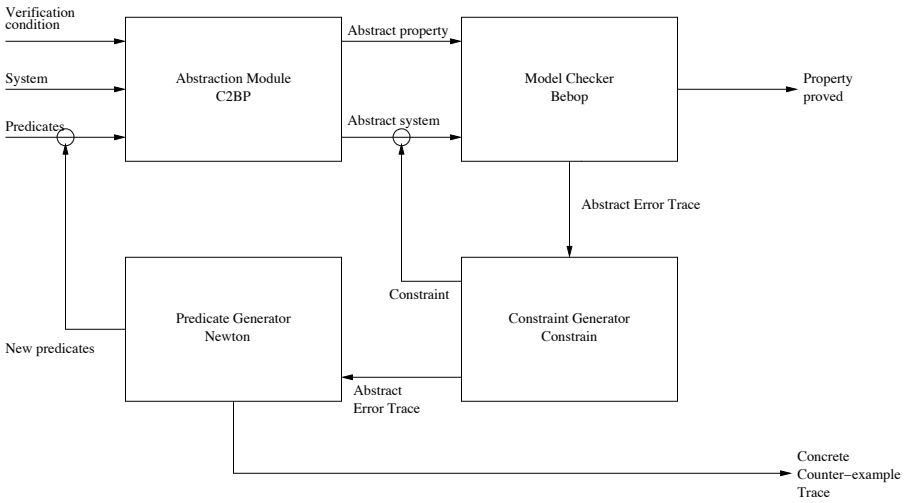


Figure 6.2: SLAM with CONSTRAIN

6.3 Approximation in C2bp

The construction of the abstract system can be done approximately while preserving correctness. The important requirement is that the constructed abstract system should have all the transitions that are allowed in the exact abstract system. When this requirement is fulfilled, the set of reachable states in the constructed abstract system is a superset of the exact abstract reachable states. Any properties that hold in the this approximation must also hold for the exact abstract system and hence in concrete system. So C2BP is allowed to construct an abstract transition relation that is an *over approximation* of the exact abstract transition relation.

Computing the approximate abstract transition relation is easier than exact abstract transition relation. Typically, the abstraction program tries to compute the exact set of abstract successors for the states that are deemed important. For the rest of the states it is assumed that they can transition to any other abstract state. In such a scenario no satisfiability tests need to be carried out for these less important states. In the limit, C2BP might consider that none of the states are important, that is the approximate abstract transition relation allows any abstract state to be the successor of any other abstract state. This is a correctness preserving approximation, but one which does not allow any properties to be checked.

```
 1)   void main()
 2)   {
 3)      int x,y,z;
 4)      if (x<y) {
 5)         if (y<z) {
 6)            if (!(x<z)) {
 7)               error();
 8)            }
 9)         }
10)      }
11)   }
```

Figure 6.3: Example

Too coarse an approximation causes SLAM to fail in the following manner:

- The approximate abstract program has an error trace, t.
- Error trace t is found to be infeasible in the concrete program, and a set of predicates to rule out the trace are generated.
- Even after the new predicates are used, SLAM fails to eliminate the infeasible error trace t due to lack of precision of C2BP.
- The SLAM tool gets stuck in loop where the same predicates are generated repeatedly. This condition is called an *NDF-state*, where NDF is an acronym for No Difference Found.

The above scenario happens frequently in practice and limits the usefulness of predicate abstraction.

Now various approximations used in C2BP are discussed.

6.3.1 Cartesian Approximation

The *cartesian approximation*[1] of the boolean abstraction is computed by disregarding the correlation between the predicates when constructing the abstract next state functions.

[1]The initial approximation of the abstract system composed in Chapter 4 is also a cartesian abstraction.

To understand the problem consider the example in Figure 6.3. The desired verification condition is that line 7 is unreachable. Assume that the three predicates, `b1` $\triangleq$ `x < y`, `b2` $\triangleq$ `y < z` and `b3` $\triangleq$ `x < z` are used for the abstraction. The corresponding abstract boolean program is shown in Figure 6.4. Notice that in line 4 of the boolean program, all the abstract variables have been set to non-deterministic values. This is a consequence of cartesian approximation, since the correlation between the predicates was not taken into account when constructing the abstract transition. As a result, line 8 in the boolean program, corresponding to the error condition in the C program, is reachable.

This issue would have been avoided if C2BP had translated the assignment as:

```
b1,b2,b3 = *,*,((b1 && b2)? 1 : *);
```

Here the correlation between the predicates is taken into account. With this statement the verification condition would have been proved.

Another approach would be to add a constraint to the boolean program that will recapture some of the accuracy that was lost by the cartesian approximation. Such a refined boolean program is shown in Figure 6.5. The `constrain` statement adds a constraint that must be satisfied by the preceding assignment statement. In the constraint, primed variables refer to the values of the variables after the assignment while unprimed variables refer to the values of the variables before the assignment. In this example, `b1`, `b2` and `b3` are assigned values such that at least one of the expressions, `b1`, `b2` or `!b3` must be false after the statement has executed. The condition of one of the if statements must fail to hold, thereby making the error statement, on line 10 of the refined program, unreachable.

The cartesian approximation is used since it requires fewer validity checks to create the abstract system. For a vast majority of real examples the approximate system produced is sufficient in proving the verification condition. But in a few cases, as seen for instance in the above example, the cartesian approximation fails. In such instances, a more exact abstract system has to be created.

```
 1)   void main()
 2)   {
 3)      bool b1,b2,b3;
 4)      b1,b2,b3 = *,*,*;
 5)      if (b1) {
 6)         if (b2) {
 7)            if (!b3) {
 8)               error();
 9)            }
10)         }
11)      }
12)   }
```

Figure 6.4: Cartesian Abstraction of Program in Figure 6.3

6.3.2 Maximum Cube Length Approximation

The maximum cube length approximation limits the number of predicates that are considered when constructing the abstract boolean program. Assume that there are eight interdependent predicates, that is predicates that depend on the same set of concrete state variables. Also let the maximum cube length considered be three. Then while computing the abstract transition functions for each of the abstract state variables, only conjuncts of length three are considered. So the number of validity checks would be at most ${}^8C_3.2^3$ instead of 2^8.

Like the cartesian approximation, this method reduces the number of validity checks needed at the cost of producing a less accurate abstract system. On many examples this method produces an approximation that suffices. As with any heuristic, occasionally this method produces an inaccurate abstract system.

6.3.3 Imprecise Theorem Prover

In the C2BP program, a fast but imprecise theorem prover is used. It is imprecise in that it could fail to prove that a true formula is valid. Thus the abstract system produced is a conservative abstraction. But it could be less precise than the exact abstract system.

```
 1)   void main()
 2)   {
 3)      bool b1,b2,b3;
 4)      b1,b2,b3 = *,*,*;
 5)      constrain(NOT (b1' && b2' && (NOT b3')));
 6)
 7)      if (b1) {
 8)         if (b2) {
 9)            if (!b3) {
10)               error();
11)            }
12)         }
13)      }
14)   }
```

Figure 6.5: Refined Abstraction of Program in Figure 6.4

6.4 Refining the abstract transition system

The imprecision introduced by C2BP is alleviated by adapting the algorithm described in Chapter 4. A constraint generator module, called CONSTRAIN is added to the SLAM tool as shown in Figure 6.2. After the possibly imprecise abstract system has been model checked and an abstract counter-example trace produced, every transition in the trace is checked to see if it would be allowed in the exact system. If a transition is found that does not satisfy this condition, the abstract transition relation is made more precise and the model checker run again.

The first step in the SLAM process is to convert C programs into an intermediate representation where all statements are either assignment statements, assume statements, function calls or function returns. For each of these statement types a function `Spurious` is defined. This function takes as its arguments, an instruction in the intermediate C description, three abstract states, s_0 and s_1 and s_c. It returns a logic formula which is valid if and only if the abstract state s_1 can not be the abstract successor of s_0. When the spuriousness of a return statement is being checked the abstract state at the call site, s_c, is also needed, since the return statement updates the predicates local to the calling context.

In this chapter, $\mathcal{E}$ is used to denote the concretization function which maps abstract states to concrete states. The pre-image of concrete states, Θ and instruction τ is given by,

$$\texttt{Pre}(\tau, \Theta) = \lambda c. \exists c'. \tau(c, c') \wedge \Theta(c')$$

For deterministic instructions, `Pre` can be computed using `WP`, the weakest precondition. Within `WP` Morris's general axiom of assignment is used to model pointer aliasing conservatively.

Using this definition of `Pre`, the `Spurious` function for statements of various types are as follows.

Assignment and Assume statements. If τ is an assignment or assume statement, then `Spurious` is defined as:

$$\texttt{Spurious}(\tau, s_0, s_1, s_c) \quad = \quad (\mathcal{E}(s_0) \Longrightarrow \neg \texttt{Pre}(\tau, \mathcal{E}(s_1))$$

Function calls and returns. Let τ be a function call to procedure R from some procedure Q in the C program:

$$x = R(e_1, e_2, \ldots, e_N);$$

Assume that the formal parameters of R be $f_1, f_2, \ldots f_N$. In the intermediate representation, the call to the function and the return from the function are explicitly made different by adding a fresh temporary variable, x_{ret} to the program:

$$x_{ret} = R(e_1, e_2, \ldots, e_N);$$
$$x = x_{ret};$$

In the boolean program, the abstract version of the function, R_A is called. Then when the concrete function returns, the abstract version returns and the values of the predicates in the calling context are updated. So the boolean program corresponding

to the above fragment is:

$$p_1, p_2, \ldots, p_M = F(ret_1, ret_2, \ldots, ret_L);$$
$$ret_1, ret_2, \ldots, ret_L = R_A(p_1, p_2, \ldots, p_M);$$
$$b_1, b_2, \ldots b_n = G(ret_1, ret_2, \ldots, ret_L);$$

In the above fragment, a conservative approximation of the parameters passed into R_A is computed in the first line. Then the abstract version of the function is called in the second line and finally the variables in the calling context are updated in the last line.

In any abstract counter-example trace there would be a transition from the calling context into the context of R_A and also a transition back. If the first transition is spurious then the set of parameters computed on the first line is too approximate. On the other hand if the transition where the function returns is spurious then accuracy was lost while computing the effects on the abstract variables in the last line.

Let τ_p denote the parallel assignment of the actuals to the formals, that is: $f_1 = e_1, f_2 = e_2, \ldots, f_N = e_N$. For the function call itself, `Spurious` is defined as:

$$\texttt{Spurious}(\tau, s_0, s_1, s_c) \quad = \quad \mathcal{E}(s_0) \Longrightarrow \texttt{Pre}(\tau_p, \mathcal{E}(s_1))$$

If the above logic formula is valid and the function δ maps formal parameter predicates to the corresponding temporaries in $\{p_1, p_2, \ldots p_N\}$, then CONSTRAIN adds $\neg(s_0 \wedge \delta(s_1'))$ as an constraint to the first line of the abstract program above.

Now the `Spurious` function is defined for function return statements. Let τ be the return statement, `return r` in the function R. Also the return value is assigned to variable x in the calling context. First two auxiliary functions: ρ over the variables in scope in the calling context and γ over the variables in scope in R are defined as follows:

$$\rho(v) = \begin{cases} v & \text{if } v \text{ is a local in } Q \\ v_0 & \text{if } v \text{ is global and } v_0 \text{ is a fresh variable} \end{cases}$$

The fresh variable, v_0 caches the value of the global variable v just before the call to R.

$$\gamma(v) = \begin{cases} v & \text{if } v \text{ is global} \\ x_{ret} & \text{if } v \text{ is } r \\ \rho(e_i) & \text{if } v \text{ is the symbolic value of } f_i \end{cases}$$

Let τ' denote the assignment statement, $x = x_{ret}$. Then:

$$\texttt{Spurious}(\tau, s_0, s_1, s_c) \quad = \quad \gamma(\mathcal{E}(s_0)) \wedge \rho(\mathcal{E}(s_c)) \Longrightarrow \neg \texttt{Pre}(\tau', \mathcal{E}(s_1))$$

If the above formula is valid then the constraint, $\neg(s_0 \wedge \omega(s_c) \wedge s_1')$ is added to the statement in the abstraction that models the side-effects of the call. The function, ω maps each predicate in R to the corresponding return temporary.

6.5 Optimizations

The optimizations used are very similar to the ones discussed in Chapter 4. The constraint added is minimized to generate a stronger constraint and support for finding invariant relationship between predicates is present.

After a spurious transition is found, terms are dropped from it in sequence and the remainder tested for spuriousness. At the end this produces a strengthened constraint. This optimization allows the program to eliminate more spurious transitions than the weaker constraint does.

If any of the states in the abstract error trace is unsatisfiable, that is there are no concrete states corresponding to it, then the CONSTRAIN program has discovered a new invariant. The boolean program description language has the `enforce` construct that can be used to tell the model checker about invariants. During model checking states that do not satisfy these invariants are never explored.

6.6 Example

The steps that CONSTRAIN follows are now described with the example described in Figure 6.3. Since cartesian approximation is used, C2BP produces the boolean program shown in Figure 6.4. The model checking fails to prove the that the error statement is unreachable. In the abstract counter-example proved there is a transition from some arbitrary initial state to a state where `b1` and `b2` are true while `b3` is false.

$$\texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3} \stackrel{5}{\Longrightarrow} \texttt{b1} \wedge \texttt{b2} \wedge !\texttt{b3}$$

Now CONSTRAIN is used to show that the transition is not allowed. Notice that the initialization of the variables `x`, `y` and `z` is equivalent to the parallel assignment,

```
x := x0, y := y0, z := z0
```

where `x0`, `y0` and `z0` are unrelated fresh constants. So from definition of `Spurious` it follows that

$$\begin{aligned} & \texttt{Spurious}(5, \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3}, \texttt{b1} \wedge \texttt{b2} \wedge \neg\texttt{b3}) \\ \equiv\ & \mathcal{E}(\texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3}) \Longrightarrow \neg\texttt{Pre}(5, \mathcal{E}(\texttt{b1} \wedge \texttt{b2} \wedge \neg\texttt{b3})) \\ \equiv\ & ((x < y) \wedge (y < z) \wedge (x < z)) \Longrightarrow \neg((x0 < y0) \wedge (y0 < z0) \wedge \neg(x0 < z0)) \end{aligned}$$

Since the above formula is valid, the transition is spurious and a constraint needs to be added to refine the boolean program. The added constraint is

```
constrain((!b1 || !b2 || !b3) || !(b1' && b2' && !b3'));
```

However some of the disjuncts above can be dropped without making the remainder invalid. The minimal formula which is still valid is

$$\neg((x0 < y0) \wedge (y0 < z0) \wedge \neg(x0 < z0))$$

The constraint corresponding to the above is

```
constrain(!(b1' && b2' && !b3'));
```

This constraint is better than the previous one since it eliminates more spurious transitions. After this constraint is added, the model checker is able to prove that the verification holds in the refined boolean program.

6.7 Experimental results

The use of CONSTRAIN makes SLAM more effective in verifying device drivers. To show how CONSTRAIN fares the results of running SLAM with the following schemes are shown:

No Constrain : SLAM without CONSTRAIN

Lazy mode : CONSTRAIN used as NDF-state recovery method. Each call to CONSTRAIN allowed to generate at most five constraints and constrain is called at most four times before returning to the usual SLAM loop.

Eager mode : SLAM with CONSTRAIN where C2BP generates an abstract transition relation that is completely unconstrained.

With each of these configurations, SLAM was applied to verify 35 safety properties of 26 separate device drivers, chosen from Windows NT. For each run of SLAM, the timeout threshold was set at 1200 seconds and the memory threshold was set at 500 MB. Drivers size ranged from 10K to 40K lines of code. The results are shown in Figure 6.1.

The sum total of the first two rows is the number of test cases where SLAM terminated with a definite result. Clearly using CONSTRAIN allows SLAM to do better. In particular the number of drivers where the SLAM terminated with NDF-state has been reduced. The three remaining cases in which SLAM still reached an NDF-state are due to some subtle problems that occur when refining abstractions in the presence of heap-based data structures. The number of drivers where the threshold is exceeded increases since SLAM could now do more analysis.

Result	No Constrain	Lazy mode	Eager mode
Property passes	470	554	544
Property violation	19	50	45
Termination in NDF-state	170	3	3
Time/memory threshold exceeded	55	107	122
Property not applicable to driver	266	266	266

Table 6.1: Results from experiments with CONSTRAIN on 26 device drivers and 35 safety properties. The time threshold was set to 1200 seconds. The memory threshold was set to 500 megabytes.

As expected in **Eager mode** SLAM does not perform as well as in **Lazy mode**. Since the initial approximation is much coarser in **Eager mode**, some of the drivers run out of time/memory before the proof can be completed.

6.7.1 Effect on performance

To quantify the cost that adding CONSTRAIN to the SLAM tool-chain, the time spent in CONSTRAIN was measured. It turns out that in **Lazy mode** 10% of SLAM's overall runtime was spent in CONSTRAIN.

In many of the examples CONSTRAIN was not run at all. So to get a better idea of the impact of using CONSTRAIN, the averages for only the 126 drivers where CONSTRAIN was needed to eliminate a NDF-state condition are presented in Figure 6.2. The effect of using CONSTRAIN is not overwhelming. It consumes about 23% of

Description	Average
SLAM runtime	354.31s
Amount of runtime spent in CONSTRAIN	81.24s
Calls to CONSTRAIN	4.40
CONSTRAIN refinement iterations	1.74
Constraints generated	6.17

Table 6.2: Averages collected when applying SLAM to 126 device-driver based benchmarks in which CONSTRAIN is required to recover from an NDF-state.

the total run time, getting called about 5 times in every model checking run and adding about 7 constraints to the final abstract transition relation. Also on average, CONSTRAIN gets called in less than two of the iterations of SLAM's main loop.

6.8 Conclusions

The techniques developed in Chapter 4 has been successfully applied to program analysis. When CONSTRAIN is used in **Lazy mode**, SLAM is able to compute a definite result for 23% more properties. We are also investigating the use of our predicate discovery algorithms in SLAM.

Chapter 7

Conclusions

Novel techniques that ease the verification of safety properties, and in many cases makes it completely automatic, has been presented in this dissertation.

7.1 Future Work

It would be interesting to develop techniques to apply predicate abstraction to prove liveness properties. To prove progress properties, a liberal approximation of the state space needs to be computed. That is we need to prove that a set of states have definitely been seen.

Better support for quantifiers in CVC would help in making predicate abstraction more efficient, especially for parameterized systems. Currently, quantifier instantiation heuristics are implemented in the predicate abstraction tool. If these are moved into the validity checker, then overall efficiency would be improved.

Integration with other invariant generation methods would be helpful. For some of the protocols that we have looked at, linear invariants are needed for verification. The predicate discovery algorithm cannot find the atomic predicates present in these linear invariants. In such cases, it would be helpful if integration with other tools is possible.

Bibliography

[1] David L. Dill Aaron Stump, Clark W. Barrett. CVC: a cooperating validity checker. In *Conference on Computer Aided Verification*, Lecture notes in Computer Science. Springer-Verlag, 2002.

[2] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation 118(1)*, pages 142–157, 1995.

[3] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *5th International Conference on Computer-Aided Verification*, pages 29–40. Springer-Verlag, 1993.

[4] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE 01: Workshop on Program Analysis for Software Tools and Engineering*, pages 97–103. ACM, 2001.

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

[6] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the Conference on Programming Languages Design and Implementation*, pages 203–213, 2001.

[7] 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.

[8] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In *10th International Conference on Computer-Aided Verification*, pages 505–510. Springer-Verlag, 1998.

[9] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols, August 1999. Presented in the Recent Research Session at Sigcomm 1999.

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

[11] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on VLSI*, August 1991.

[12] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. In *International Conference on Software Engineering*, pages 385–395, May 2003.

[13] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169. Springer-Verlag, 2000.

[14] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[15] Michael A. Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Conference on Computer-Aided*

Verification, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, 1998.

[16] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Symposium on the Principle of Programming Languages*, pages 238–252, 1977.

[17] E. W. Dijkstra, L. Lamport, A.J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–75, November 1978.

[18] David L. Dill. The Murϕ verification system. In *Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, pages 390–393. Springer-Verlag, July 1996.

[19] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2002.

[20] J. A. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. In *Proc. Advances in Cryptology - Crypto '99*, pages 449–466, 1999.

[21] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Conference on Computer Aided Verification*, volume 1254 of *Lecture notes in Computer Science*, pages 72–83. Springer-Verlag, 1997. June 1997, Haifa, Israel.

[22] Klaus Havelund. Mechanical verification of a garbage collector. In José Rolim et al., editors, *Parallel and Distributed Processing (Combined Proceedings of 11 Workshops)*, volume 1586, pages 1258–1283, San Juan, Puerto Rico, 1999. Springer-Verlag.

[23] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In

P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 54–69, Liege, Belgium, 1995. Springer Verlag.

[24] Henrik Jensen and Nancy Lynch. A proof of burns n-process mutual exclusion algorithm using abstraction. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 409–423, 1998.

[25] Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, pages 98–112, Genova, Italy, 2001. Springer-Verlag.

[26] D. Lessens and Hassen Saïdi. Automatic verification of parameterized networks of processes by abstraction. *Electronic Notes of Theoretical Computer Science (ENTCS)*, 1997.

[27] Nancy Lynch and Mark Tutle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[28] Zohar Manna and Amir Pnueli. *Tempral Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

[29] Seungjoon Park and David L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.

[30] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on-demand distance vector (aodv) routing. In *Workshop on Mobile Computing Systems and Applications*, pages 90–100. ACM Press, February 1999.

[31] Charles E. Perkins, Elizabeth M. Royer, and Samir Das. Ad hoc on-demand distance vector (aodv) routing. Available at `http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-05.txt`, 2000.

[32] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In *Computer Aided Verification (CAV)*, 1996.

[33] Hassen Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering (ASE '99)*, 1999.

[34] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In *11th International Conference on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

[35] V. Shmatikov and J. C. Mitchell. Analysis of abuse-free contract signing. In *Financial Cryptography*, 2000. Anguilla.

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

[37] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In Tiziana Margaria and Wang Yi, editors, *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, apr 2001. Springer-Verlag.

[38] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using bdds. In *Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.