SAT-Based Predicate Abstraction of Programs

Edmund Clarke
Carnegie Mellon University,
School of Computer Science
Daniel Kroening
Swiss Federal Institute of Technology, Zurich
Natasha Sharygina
Carnegie Mellon University,
Software Engineering Institute,
School of Computer Science
Karen Yorav
IBM, Haifa, Israel

September 2005

Predictable Assembly from Certifiable Components Initiative

Technical Report
CMU/SEI-2005-TR-006
ESC-TR-2005-006

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Abstract ..... **v**

1 Introduction **1**

2 Model Checking **5**
2.1 The Process of model checking 6
2.2 Current Research in Software Model Checking 6
2.2.1 Abstraction 7
2.2.2 Counterexample-Guided Abstraction Refinement (CEGAR) 7

3 SAT-Based Abstraction for Software **11**
3.1 Motivation 11
3.2 A Boolean Formula for the Concrete Transition Relation 12
3.2.1 Preparation 12
3.2.2 Translating Assignments into Bit-Vector Equations 13
3.2.3 Programs That Use Pointers 15
3.2.4 Translating Bit-Vector Equations into Boolean Formulas 16
3.3 Using SAT to Compute the Abstraction 17
3.3.1 The Abstract Transition Relation for a Basic Block 17
3.3.2 The Abstract Transition Relation for Control-Flow Statements 19
3.4 The Implementation 20
3.4.1 Minimizing the Number of Quantified Variables 20
3.4.2 Obtaining the Set of Satisfying Assignments 21
3.4.3 Using SMV to Check the Abstract Program 21
3.4.4 Simulating the Abstract Counterexample 22
3.4.5 Verifying Properties of the Program 23
3.5 Experimental Results 24
3.5.1 Secure Hash Algorithm (SHA) 24
3.5.2 ASN1 Data Structures in OpenSSL 24
3.5.3 MD2 Message-Digest Algorithm 26
3.5.4 Pointer Arithmetic in JPEG Decoder 26
3.6 Related Work 26

4 Conclusion **31**

References **33**

List of Figures

Figure 1: Applying the ComFoRT Reasoning Framework 2
Figure 2: The CEGAR Framework 8
Figure 3: Translation of a Basic Block into Its Single Assignment Form 13
Figure 4: Example: Generation of the Concrete Transition Relation 17
Figure 5: Excerpt from an SHA Implementation 25
Figure 6: Excerpt from an Implementation of ASN1 Data Structures from OpenSSL . 25
Figure 7: Excerpt from the Reference Implementation of the MD2 Algorithm 27
Figure 8: Excerpt from JPEG decoder 28

Abstract

Component Formal Reasoning Technology, ComFoRT, is a model-checking-based approach for analysis of component-based software designs. ComFoRT is designed to be used in a prediction-enabled component technology (PECT). A PECT provides a means to reliably predict the runtime qualities (e.g., performance and reliability) of assemblies of components from their certifiable properties (e.g., execution time and behavioral descriptions). ComFoRT uses an abstraction-based approach to cope with the complexity of analysis by reducing the size of the program models to be analyzed. This note presents technical details of a SAT-based predicate abstraction technique used in ComFoRT. The main advantage of the SAT-based method over conventional predicate abstraction techniques is that it does not require an exponential number of theorem prover calls for computing an abstract model. Additionally, the SAT-based approach computes a more precise and safe abstraction compared to existing predicate abstraction methods.

1 Introduction

Component Formal Reasoning Technology, ComFoRT, is a result of the Predictable Assembly from Certifiable Components (PACC) Initiative at the Carnegie Mellon® Software Engineering Institute (SEI), which provides a means to reliably predict the runtime qualities (e.g., performance or reliability) of component assemblies from their certifiable properties (e.g., execution time or behavioral descriptions).

The underlying formal analysis technique of ComFoRT is model checking—an automated algorithmic approach for exhaustively analyzing whether concurrent finite-state models satisfy specific behavioral claims. The types of claims evaluated are temporal expressions over system execution; for example, checking whether a system can ever fail to answer a message while in a normal mode of operation or deadlock. Checking these types of claims allows developers to determine whether systems will respond correctly and satisfy specific safety and reliability requirements under all possible conditions.

While model checking has been successfully applied to software, model checking successes are far more common in hardware industry applications and in research settings than in industrial software development. Two causes of the dearth of software successes are (1) the theoretical problems limiting successful application to software (for example, state space explosion) and (2) the fact that commercial developers typically lack the theoretical expertise needed to apply model checking. The PACC approach to addressing these problems is to incorporate state-of-the-art model checking into a prediction-enabled component technology (PECT) [Wallnau 03a].

A PECT packages the complexities of an analysis technology (for example, model checking) in a reasoning framework that is combined with a component technology. That combination allows developers to predict the behavior of their component-based systems without having to become experts in the analysis technology. Reasoning frameworks also exploit a component technology's features and constraints to scope the class of designs that must be considered. Such scoping can alleviate theoretical problems and improve applicability.

Developers use the construction framework of a PECT to describe the component assemblies. The construction framework includes a language for describing components and their assemblies (the construction and composition language [CCL] [Wallnau 03b])† and tools for designing, developing, and deploying components and their assemblies based on their CCL specifications [Hissam 02].

One of many important tasks performed by these tools is constraint checking; each CCL specification is checked to ensure that it satisfies constraints imposed by the component technology and the reasoning framework. This step ensures that any system that can be built using the PECT can also be analyzed, which leads to systems that are *predictable by construction*.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

† We do not restrict ourselves to CCL even though it is currently used in our project. CCL will be broadened in the future with more design and construction notations.

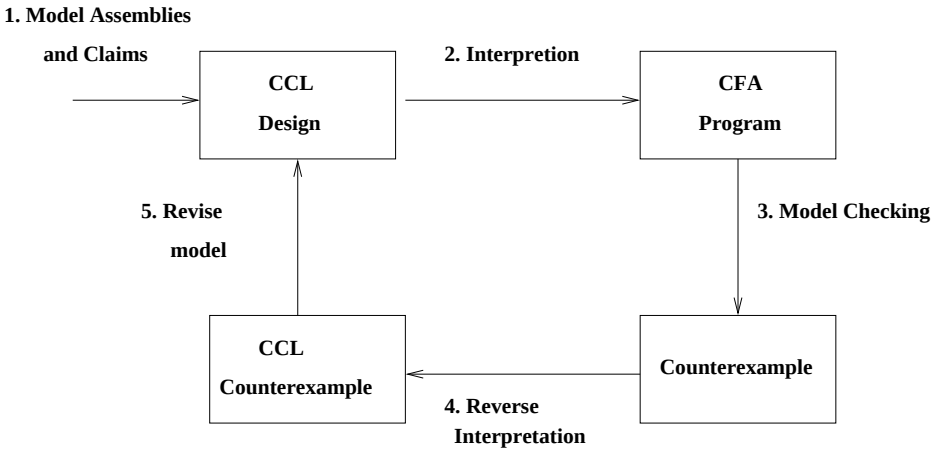


Figure 1: Applying the ComFoRT Reasoning Framework

Each reasoning framework is applied to assembly specifications by means of a formal interpretation that generates reasoning-framework-specific models from CCL specifications. Figure 1 shows a general overview of how the ComFoRT framework is used in the PECT environment. PECT users design components and assemblies using CCL as their design language and add property annotations as needed to enable specific predictions (i.e., to satisfy specific reasoning framework constraints). PECT automation handles the rest: it generates reasoning-framework-specific models (CFA programs that are essentially a mix of C and FSP code) by interpretation, uses a reasoning framework to compute predicted behavior, and translates the results back into concepts from the original CCL specifications.

Overall, the ComFoRT reasoning framework employs the following technologies:

- model checking
- automated predicate abstraction
- automated abstraction refinement (also known as counterexample-guided abstraction refinement [CEGAR])
- SAT-based predicate abstraction
- compositional reasoning
- automated component substitutability analysis
- hybrid state/event system modeling and specification (state/event linear time logic (LTL) and universal subset of branching time logic [ACTL])

In this document, we describe a particular model checking approach, namely a SAT-based predicate abstraction technique, that enables efficient and accurate modeling and verification of software. Details of the other techniques can be found in the ComFoRT overview [Ivers 04] and descriptions of the ComFoRT predicate abstraction using decision procedures for the

integer domain and the CEGAR loop [Clarke 04a], compositional reasoning and component substitutability checks [Chaki 04], and state/event notation [Clarke 04c, Clarke 04b].

The rest of this document is organized as follows: Section 2 provides some background information on the model checking technology, the types of claims it can analyze, and the current state of the research and practice of model checking. Sections 3 describes the SAT-based model checking approach and its valuation during verification of real-life examples. Section 4 summarizes the advantages of the SAT-based predicate abstraction approach.

2 Model Checking

In formal verification, a system is modeled mathematically, and its specification (also called a claim in model checking) is described in a formal language. When the behavior in a system model does not violate the behavior specified in a claim, the model satisfies the specification. *model checking* [Clarke 82] is a fully automated form of formal verification that uses algorithms that check whether a system satisfies a desired claim through an exhaustive search of all possible executions of the system. The exhaustive nature of model checking renders the typical testing question of adequate coverage unnecessary.

Model checking is a technique for verifying finite-state concurrent systems. One benefit of restricting ourselves to finite-state systems is that verification can be performed automatically. Given sufficient resources, model checking always terminates with a yes or no answer. Moreover, it can be implemented by algorithms that have reasonable efficiency and that can be run on moderate-sized machines.

Although the restriction to finite-state systems may seem to be a major disadvantage, model checking is applicable to several very important classes of systems [Clarke 99]. Hardware controllers are finite-state systems, as are many communication protocols. Software, which is not finite state, may still be verified if variables are assumed to be defined over finite domains. This assumption does not restrict the applicability of model checking, since many interesting behaviors of the software systems can be specified with finite-state models. For example, systems with unbounded message queues can be verified by restricting the size of the queues to a small number like two or three.

In classical model checking, systems are modeled mathematically as state transition systems and claims are specified using temporal logic [Pnueli 77, Clarke 86]. Temporal logic is used to define formulas that describe system behavior over time, where the propositions of the logic are behaviors of interest involving state information (current state or values of variables) or events. Temporal logic formulas combine such propositions with temporal operators to describe interesting patterns of propositions over time, such as

- Whenever X is greater than Y, Z must also be greater than Y.
- Some invariant (e.g., mutual exclusion with respect to some resource) always holds once initialization is complete.
- A component can only issue requests during an allowed interval (as bounded by events granting and taking away permission).

Temporal logic model checking is extremely useful in verifying the behavior of systems composed of concurrent processes or interacting nondeterministic sequential tasks. Concurrency errors (as well as errors caused by the nondeterministic execution of actions) are among the most difficult to find by testing, since they tend to be nonreproducible.

2.1 The Process of model checking

model checking involves the following steps:

1. The system is modeled using the description language of a model checker, producing a model M.
2. The claim to check is defined using the specification language of the model checker, producing a temporal logic formula ϕ.
3. The model checker automatically checks whether $M \models \phi$ satisfies.

The model checker checks all system executions captured by the model and outputs the answer yes if the claim holds in the model (M) and the answer no otherwise. When a claim is not satisfied, most model checkers produce a *counterexample* of system behavior that causes the failure. A counterexample defines an execution trace that violates the claim. Counterexamples are one of the most useful features of model checking, as they allow users to quickly understand why a claim is not satisfied.

2.2 Current Research in Software Model Checking

Model checking is efficient in hardware verification, but applying it to software is complicated by several factors, ranging from the difficulty of modeling computer systems (due to the complexity of programming languages as compared to hardware description languages) to difficulties in specifying meaningful claims for software using the usual temporal logical formalisms of model checking. The most significant limitation, however, is the *state space explosion problem* (which applies to both hardware and software), whereby the complexity of model checking becomes prohibitive.

State space explosion results from the fact that the size of the state transition system is exponential in the number of variables and concurrent units in the system. When the system is composed of several concurrent units, its combined description may lead to an exponential blowup as well. The state space explosion problem is the subject of most model checking research.

The ComFoRT reasoning framework exploits current state-of-the-art research efforts in the model checking community. To combat state space explosion, it applies the following major state-space-reduction techniques:

- *compositional reasoning.* Verification is partitioned into checks of individual modules, while the global correctness of the composed system is established by constructing a proof outline that exploits the modular structure of the system.
- *abstraction.* A smaller abstract system is constructed such that the claim holds for the original system if it holds for the abstract system.
- *CEGAR.* Abstracted systems are refined iteratively using information extracted from counterexamples until an error is found or it is proven that the system satisfies the verification claim.

This technical report is concerned with the abstraction techniques. Ivers and Sharygina discuss the other approaches used in ComFoRT in detail [Ivers 04].

2.2.1 Abstraction

Abstraction is one of the principal complexity reduction techniques [Ball 01a, Clarke 92, Kurshan 95]. Abstraction techniques reduce the state space by mapping the concrete set of actual system states to an abstract set of states that preserve the actual system's behavior. Abstractions are usually performed in an informal, manual manner and require considerable expertise. Predicate abstraction [Graf 97, Colón 98] is one of the most popular and widely applied methods for the systematic abstraction of systems. It maps concrete data types to abstract data types through predicates over the concrete data. However, the computational cost of the predicate abstraction procedure may be too high, making generation of a full set of predicates for a large system infeasible.

In practice, the number of computed predicates is bounded, and model checking is guaranteed to deliver sound results within this bound. The bound limit is increased once errors (if any) are found within the bound and fixed. Under this approach, software systems are rendered finite by restricting variables to finite domains. As mentioned earlier, bounded model checking does not seriously restrict the applicability of model checking, since many interesting behaviors of software systems can be specified using bounded finite-state models.

The abstract program is created using *existential abstraction* [Clarke 92]. This method defines the transition relation of the abstract program, so it is guaranteed to be a *conservative* overapproximation of the original program, with respect to the set of given predicates. The use of a conservative abstraction, as opposed to an *exact* abstraction, produces considerable reductions in the state space. The drawback of the conservative abstraction is that when model checking of the abstract program fails, it may produce a counterexample that does not correspond to a concrete counterexample. Such a counterexample is usually called *spurious*. When a spurious counterexample is encountered, *refinement* is performed by adjusting the set of predicates in a way that eliminates it.

2.2.2 Counterexample-Guided Abstraction Refinement (CEGAR)

Though conservative abstraction procedures—which ensure that if a claim holds for the abstract system, it also holds for the original system—are typically used, any form of abstraction may introduce behaviors not found in the concrete system. Counterexamples from model checking the abstract system are often used to detect unrealistic behaviors and refine the system. Repeatedly refining the abstractions, however, may introduce additional behaviors that result in state space explosion during the model checking phase. These drawbacks—coupled with the potential effectiveness of abstraction methods—motivate research into targeted abstractions (i.e., control abstraction, loop abstraction, and so forth), which can result in more accurate abstract systems.

The abstraction refinement process has been automated by the CEGAR paradigm [Kurshan 95, Ball 00, Clarke 00, Das 01]. The CEGAR framework is shown in Figure 2: you start with a coarse abstraction (for example, an abstraction of a C program). If an error trace

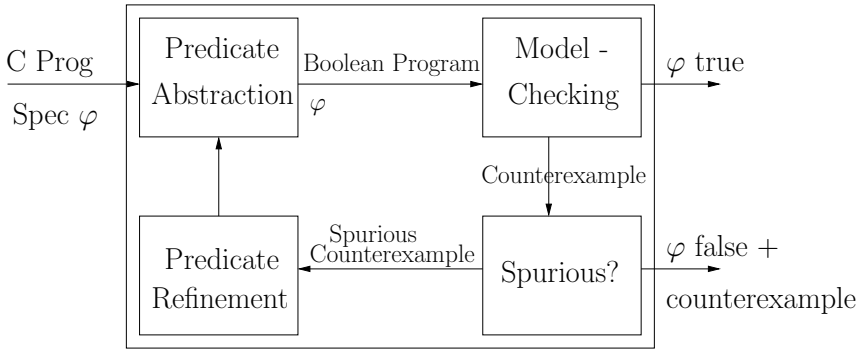


Figure 2: The CEGAR Framework

reported by the model checker is not realistic, the trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the *abstract-verify-refine* paradigm and depend on the abstraction and refinement techniques used. The steps are described below in the context of predicate abstraction.

1. **program abstraction.** Given a set of predicates, a finite-state model is extracted from the code of a software system, and the abstract transition system is constructed.

2. **verification.** A model checking algorithm is run to check whether the model created by applying predicate abstraction satisfies the desired behavioral claim φ. If the claim holds, the model checker reports success (φ is *true*), and the CEGAR loop terminates. Otherwise, the model checker extracts a counterexample, and the computation proceeds to the next step.

3. **counterexample validation.** The counterexample is examined to determine whether it is spurious. This examination is done by simulating the (concrete) program using the abstract counterexample as a guide, to find out if the counterexample represents an actual program behavior. If this is the case, the bug is reported (φ is *false*), and the CEGAR loop terminates. Otherwise, the CEGAR loop proceeds to the next step.

4. **predicate refinement.** The set of predicates is changed to eliminate the detected spurious counterexample and possibly other spurious behaviors introduced by predicate abstraction. Given the updated set of predicates, the CEGAR loop proceeds to Step 1.

The efficiency of this process is dependent on the efficiency of the *program abstraction* and *predicate refinement* procedures. While program abstraction focuses on constructing the transition relation of the abstract program, the focus of predicate refinement is to define efficient techniques for choosing the set of predicates in a way that eliminates spurious counterexamples. In both areas of research, low computational cost is a key factor, since it enables the application of model checking to the verification of realistic programs.

This report presents a technique that enables efficient abstraction (Step 1 of the CEGAR loop) of a program by using a SAT solver to generate the abstract transition relation. This technique was originally described by Clarke and colleagues [Clarke 04e].

3 SAT-Based Abstraction for Software

3.1 Motivation

In the existing software verification tools such as SLAM [Ball 01b], BLAST [Henzinger 02], and MAGIC [Chaki 03a] (the latter is the original prototype of the ComFoRT model checking engine), the generation of the abstract Boolean program from a C program and a set of predicates suffer from multiple problems:

1. The generation of the Boolean program is done by calling a theorem prover for each potential assignment to the current and next state predicates. For the most precise transition relation, this operation requires an exponential number of theorem prover calls. Several heuristics are used to reduce this number. Some existing tools avoid this large number of theorem prover calls by using a user-specified maximum. After this specified number is reached, the tool adds all remaining transitions for which the theorem prover call was skipped. This is a safe overapproximation, but it will yield a potentially large number of unnecessary spurious counterexamples.

2. Existing work relies on general-purpose theorem provers. Program variables are modeled as unbounded integer values, neglecting possible arithmetic overflow in the ANSI-C program. The use of general-purpose theorem provers can result in false positive answers of the tool.

3. Existing tools only support a very limited range of operators, namely Boolean operators, addition/subtraction, equality, and relational operators. Other ANSI-C operators—such as those for multiplication and division and bit-wise and type conversion operators—and shift operators are modeled by means of uninterpreted functions. The use of such functions limits the set of programs and properties that can be verified.

4. Existing tools only provide a limited support for pointer operations. In particular, pointer arithmetic is not handled.

One of the ComFoRT model checking approaches that overcomes these limitations is the approach that uses a SAT solver. This report presents the details of using the SAT solver for program abstraction. The SAT-based abstraction technique is defined in the context of ANSI-C programs. However, the method is general and can be applied to programs written in other imperative programming languages.

Overview of the SAT-based abstraction approach. For each basic block in the given program, the SAT-based approach first constructs a symbolic representation of the concrete transition relation by applying symbolic simulation techniques (similar to that of Currie and colleagues [Currie 00]). Next, the approach adds the predicates in current and next state form to the relation between variables, resulting in a Boolean formula. Finally, the approach enumerates symbolically on the values of the predicates, using a SAT solver. When the

abstract program needs to be refined, the ComFoRT approach uses the same formula it already created, together with the new set of predicates, to create the new abstraction.

Advantages. One advantage of the SAT-based technique is that the exponential number of theorem prover calls is eliminated; instead, the possible assignments to the values of the predicates are searched by the SAT solver. Modern SAT solvers are highly efficient and allow a large number of variables. The high efficiency of SAT solvers enables checking many more possible assignments, resulting in a more precise abstract transition relation, and eliminating redundant spurious counterexamples.

Another advantage of the SAT-based approach is that, unlike most existing tools that use decision procedures for abstraction and refinement that are limited to integers, it allows handling the semantics of bit-vector overflow and pointer arithmetic overflow accurately. Thus, there are no false positive answers due to the inaccurate assumption that the range of variable values is infinite. The only limitation is that recursion and dynamic memory allocation are not allowed. This limitation cannot be avoided, since the Boolean abstract program is required to be finite.

3.2 A Boolean Formula for the Concrete Transition Relation

This section discusses the details of constructing a Boolean formula for the concrete transition relation. The program is first partitioned into basic blocks, which are sequentially composed assignments, and control-flow statements, such as, `if`, `while`, `goto`, and so on.

We use bit-vector equations to capture the semantics of assignments. Doing so implies a different approach for control-flow statements and basic blocks. Since control-flow statements do not change variable values (we remove side effects from conditions in a preprocessing step), they do not require equations. The abstraction of control statements is therefore not described here but rather deferred to Section 3.3.2. For the rest of this section, we are only concerned with basic blocks.

Section 3.2.1 describes syntactic program transformations that are required to prepare the basic block for the translation into a bit-vector equation. Section 3.2.2 gives details on how assignments are translated into bit-vector equations using symbolic simulation techniques. Section 3.2.3 explains how this is done in the presence of pointers. The translation described here is an adaptation of the method presented by Kroening, Clarke, and Yorav [Kroening 03a, Kroening 03b]. Section 3.2.4 shows the translation of the generated bit-vector equation system into a Boolean formula, which is suitable for a SAT solver.

3.2.1 Preparation

We assume that B is a basic block containing n statements $s_1, \ldots, s_n$. This code has already been manipulated to remove function calls and empty (skip) statements, so we can assume that each s_i is an assignment. We use the notation $lhs(s_i)$ and $rhs(s_i)$ for the left-hand side and right-hand side of the assignment, respectively.

Given an expression e, we use $Vars(e)$ to denote the set of variables referenced by this expression. We use this notation also for assignments so that
$Vars(s_i) = Vars(lhs(s_i)) \cup Vars(rhs(s_i))$.

We first transform B into single assignment form in which each variable is assigned a value only once. To do so, we add auxiliary variables that record intermediate values. Let v be a variable and s_i an assignment such that $v \in Vars(s_i)$. Let $\alpha(v, s_i)$ denote the number of assignments made to variable v within the basic block prior to the statement s_i. Formally

$$\begin{aligned} \alpha(v, s_1) &= 0 \\ \forall i \geq 2 : \alpha(v, s_i) &= \begin{cases} \alpha(v, s_{i-1}) + 1 & : \ s_{i-1} \text{ assigns to } v \\ \alpha(v, s_{i-1}) & : \ \text{otherwise} \end{cases} \end{aligned}$$

Definition 1 (ρ) *Let s_i be an assignment that assigns to the variable v. Then the leftmost occurrence of v in $lhs(s_i)$ is renamed to $v_{\alpha(v,s_i)+1}$. All other occurrences of v are renamed $v_{\alpha(v,s_i)}$. Any other variable $u \in Vars(s_i)$ such that $u \neq v$ is renamed $u_{\alpha(u,s_i)}$.*

Let e denote any expression (whether a part of an assignment, a whole assignment, a condition, etc). Then $\rho(e)$ denotes the expression after this renaming.

Figure 3 gives an example of a simple block and its translation.

```
x = z * x;          ρ          x1 = z0 * x0;
y = x + 1;      ------->       y1 = x1 + 1;
x = x + y;                     x2 = x1 + y1;
```

Figure 3: Translation of a Basic Block into Its Single Assignment Form

In the rest of the report, we use v for a program variable (such as `x` in the example above) and v_j for one of its renamed versions ($\mathtt{x}_0$, $\mathtt{x}_1$, $\mathtt{x}_2$ in that example).

3.2.2 Translating Assignments into Bit-Vector Equations

We next define an equation $\sigma(s_i)$ for each assignment in the block, describing the effect this assignment has on the (renamed) variables. In this section, we assume that the program does not have any pointer variables; Section 3.2.3 will extend the method to programs that manipulate pointers.

As an intermediate format, we use bit-vector equations. Besides the usual bit-wise and arithmetic operators, we also consider the array index operator [ ], the structure member operator, and the choice operator to be part of the logic. The choice operator "?" is defined as

$$c?a : b \ \triangleq \ \begin{cases} a & : \ c \neq 0 \\ b & : \ \text{otherwise} \end{cases}$$

Furthermore, we define the `with` operator [Gries 80] for arrays and structures. It is also considered part of the bit-vector logic.

Definition 2 (`with` operator for arrays) *Let g be an expression of array type, i be an integer expression, and e be an expression with the type of the elements in g. The operator* `with` *takes g, i, and e and produces an array that is identical to g, except for the content of $g[i]$ being replaced by e. Formally, let g' be "g with $[i] := e$", then*

$$g'[j] \triangleq \begin{cases} e & : \ j = i \\ g[j] & : \ otherwise \end{cases}$$

Definition 3 (`with` operator for structures) *Let s be a variable of structure type, f be a field name of this structure, and e be an expression matching the type of the field f. The operator* `with` *takes s, f, and e and produces a structure that is identical to s, except for the content of $a.f$ being replaced by e. Formally, let s' be "s with $.f := e$" and j be a field name of s, then*

$$s'.j \triangleq \begin{cases} e & : \ j = f \\ s.j & : \ otherwise \end{cases}$$

The translation of an assignment into a constraint is done using an auxiliary function $\ell(l, r)$. It maps the expressions l for the left-hand side and r for the right-hand side into a constraint. It is defined recursively on the structure of the expression l:

- If l is a symbol v, then $\ell(l, r)$ is the equality of the left-hand side l and the right-hand side r.

$$\ell(v, r) \ := \ v = r$$

- If l is an array index expression $g[i]$ with array expression g and index expression i, then $\ell(l, r)$ is applied recursively to g and a new right-hand side, which is g with element i changed to r.

$$\ell(g[i], r) \ := \ \ell(g, g \text{ with } [i] := r)$$

- If l is a structure member expression $s.f$ with structure expression s and field name f, we define $\ell(l, r)$ in analogy to the previous case:

$$\ell(s.f, r) \ := \ \ell(s, s \text{ with } .f := r)$$

Using this auxiliary function, the function $\sigma(s_i)$ is easily defined as

$$\sigma(s_i) \ := \ \ell(lhs(s_i), rhs(s_i))$$

Our final bit-vector equation is the conjunction of the constraints generated:

$$\bigwedge_{i=1,\ldots,n} \sigma(s_i)$$

As a shorthand, let v denote the version of the variable v with index 0 and v' denote the version of the variable v with the largest index, or formally

$$\begin{aligned} v &:= v_0 \\ v' &:= v_{\alpha(v,s_{n+1})} \end{aligned}$$

Note that for any variable v that is not assigned to within the basic block, v' is just another shorthand for v_0. This gives us a bit-vector equation system that defines a relation $\mathcal{T}(\overline{v}, \overline{v}')$, where $\overline{v}$ is the vector of all variables v, and $\overline{v}'$ is the vector of all variables v'. The relation is a symbolic representation of the concrete transition relation of the block B, that is, the vector $\overline{v}$ represents the state before the execution of the basic block, and $\overline{v'}$ represents the state after the execution of the basic block. Every solution to this equation system represents a possible computation of the basic block.

3.2.3 Programs That Use Pointers

While other tools rely solely on static analysis techniques to determine the set of variables a pointer may point to, we also look at the *predicates*. As will be evident in the following section, the size of the generated equation for a statement involving a pointer p is linear in the number of objects p may point to. Thus, it is desirable to keep this number small. In a typical application, a large number of variables may have the type defined as $*p$, while there could be only a few objects that p can actually point to. To minimize the size of the equation generated, we use all the information we can extract from the program about the possible targets of p. Using the (dynamic) information obtained from the predicates, we can save a lot more than by merely using static points-to algorithms.

Before giving the formal definition, we motivate our construction as follows: When a pointer p is dereferenced and the abstract state does not hold enough information to guarantee that p is a valid, active object, the abstract program must generate an exception. During the construction of the program abstract model, it is necessary to make the abstraction safe—that is, the abstract program can refrain from generating an exception only if it is guaranteed that the concrete program does not generate an exception. For example, assume that p may point to one of $\{x, y, z\}$, while the set of predicates that involve p is $\{(p = \&x), (p = \&y)\}$. The abstract program cannot distinguish between p pointing to z, or p being `NULL`, or even p pointing to some other illegal address. Whenever p is dereferenced while both predicates are false, the abstract program will generate an exception. This means that when creating the abstract transition relation we can ignore the possibility of p pointing to z and treat it in the same way as if p were `NULL`.

The concrete transition relation we generate therefore actually depends on the predicates and is already an abstraction of the concrete behavior.

Let $\Theta(p)$ denote the set of variables that p can legally point to (i.e., the variables with a compatible type). The variables in $\Theta(p)$ are variable names *before* renaming. We analyze the set of predicates $\mathcal{P}$ and extract a set of variables $\theta(p, \mathcal{P}) \subseteq \Theta(p)$, such that $v \in \theta(p, \mathcal{P})$ holds if it is possible to derive from the predicates that p points to v. This information usually comes from predicates of the form $p = \&x$, $p = \&x + i$, $p = q$, and so on.

Definition 4 ($\theta(p, \mathcal{P})$) *Let $\mathcal{P} = \{\pi_1, \ldots, \pi_k\}$ be the set of predicates. Then $\theta(p, \mathcal{P})$ is the set of variables $v \in \Theta(p)$ for which there exists a truth assignment to the predicates such that the resulting conjunction implies that p holds the address of v.*

$$\theta(p, \mathcal{P}) \stackrel{\triangle}{=} \{v \in \Theta(p) \mid \exists b_1, \ldots, b_k.(\bigwedge_{i=1,\ldots,k} (b_i \leftrightarrow \pi_i) \Rightarrow (p = \&v)\}$$

A pointer dereference `*p` in an expression is replaced by a case split on all the variables from $\theta(p, \mathcal{P})$. Let $\theta(p, \mathcal{P}) = \{v^1, \ldots, v^k\}$. We replace every occurrence of `*p` with

$$\texttt{(p==\&}v^1\texttt{) ? } v^1 \texttt{ : (p==\&}v^2\texttt{) ? } v^2 \texttt{ : ...(p==\&}v^k\texttt{) ? } v^k \texttt{ : } \bot$$

where $\bot$ is a default value that is never used. It is important to notice that $\&v^i$ (the address of v) is a constant value and does not get renamed, while v^i is a variable name and will be given an index during the renaming process ρ. The end result is that when a pointer is dereferenced in the right-hand side of an assignment, or in the index of an array on the left-hand side, the correct value will be used. Note that it is not necessary to include all variables in $\Theta(p)$, since we generate an exception if p does not point to an object in $\theta(p, \mathcal{P})$.

The case in which a pointer dereference appears on the left-hand side of an assignment is again handled by a transformation of the program, before renaming is applied. The assignment `*p = exp` is capable of affecting any variable with the correct type. We therefore replace this assignment with a series of assignments. For each variable $u \in \theta(p, \mathcal{P})$, we add an assignment of the following form:

$$u \texttt{ = (p==\&}u\texttt{) ? exp : } u\texttt{;}$$

Again, we may refrain from adding an assignment for any variable not in $\theta(p, \mathcal{P})$, since if p points to such a variable, there will be an exception.

The transformed program does not have pointer dereferences and can be translated into an equation system using the σ function presented in the previous section. Notice that for the assignment $p = \&x$, the rule for $\sigma(v_j = exp)$ applies without change. The address of a variable is treated as a value and assigned into a variable with an appropriate type.

An example of the process described above is given in Figure 4. The example gives a basic block, the renamed version, and the resulting equation system. As an optimization, we restrict the case splits done for pointers using information from the predicates. For this example, assume the predicates $p = \&x$ and $p = \&y$.

3.2.4 Translating Bit-Vector Equations into Boolean Formulas

The translation of the bit-vector logic used to build the equation for the concrete transition relation is straightforward. We build a circuit representation that is then translated into conjunctive normal form (CNF). Several optimizations can be done at this level, in particular for arrays. The result of this process is a CNF formula $\mathcal{T}(\overline{v}, \overline{v}')$ that is a symbolic representation of the concrete transition relation.

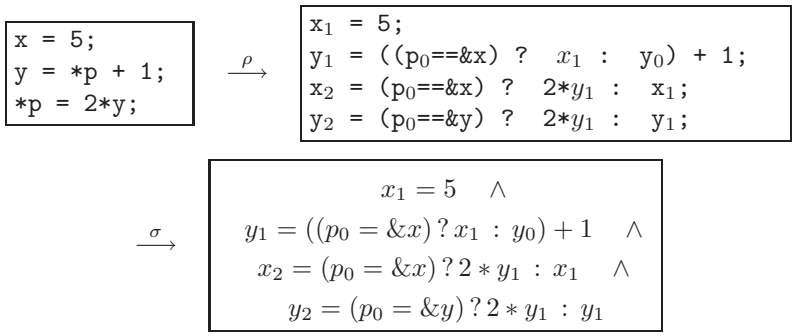


Figure 4: Example: Generation of the Concrete Transition Relation

3.3 Using SAT to Compute the Abstraction

3.3.1 The Abstract Transition Relation for a Basic Block

Let $\mathcal{P}$ be the set of predicates, where each predicate is an expression over the (concrete) program variables. Each predicate $\pi_i \in \mathcal{P}$ is associated with a Boolean variable b_i that represents its truth value. Let $\overline{\pi}$ denote the vector of predicates π_i, and $\overline{b}$ denote the vector of the Boolean variables b_i. These Boolean variables are the variables of the Boolean program we are constructing. The predicates map a concrete state $\overline{v}$ into an abstract state $\overline{b}$, and thus, $\overline{\pi}(\overline{v})$ is also called the abstraction function. Given $\mathcal{T}(\overline{v}, \overline{v}')$ and $\mathcal{P}$, we create an abstract transition relation $\mathcal{B}(\overline{b}, \overline{b}')$ that is an existential abstraction of a basic block of the C program.

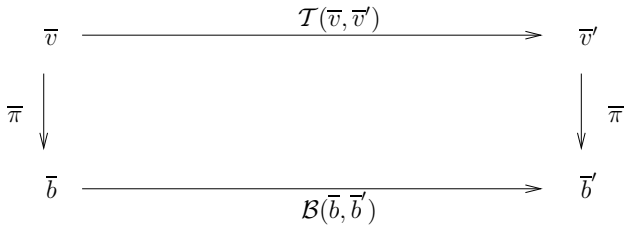
Our goal is to replace a basic block with an expression that describes what happens to the variables $\overline{b}$ when this basic block is executed. We present a translation that is accurate — that is, it produces the transition relation defined by existential abstraction. The abstract transition is not an overapproximation of the transition relation, as is commonly done by other tools.

Let $\mathcal{T}(\overline{v}, \overline{v}')$ denote the CNF formula representing the concrete transition relation, as defined in the previous section. Let $\Gamma(\overline{b}, \overline{b}', \overline{v}, \overline{v}')$ denote mapping from concrete states to abstract states. The abstract transition relation $\mathcal{B}(\overline{b}, \overline{b}')$ relates a current state $\overline{b}$ (before the execution of the basic block) to a next state $\overline{b}'$ (after the execution of the basic block). It is defined using $\overline{\pi}$ as follows:

$$\Gamma(\overline{b}, \overline{b}', \overline{v}, \overline{v}') \quad \triangleq \quad (\overline{\pi}(\overline{v}) = \overline{b}) \wedge \mathcal{T}(\overline{v}, \overline{v}') \wedge (\overline{\pi}(\overline{v}') = \overline{b}') \tag{3.1}$$

$$\mathcal{B}(\overline{b}, \overline{b}') \quad \Longleftrightarrow \quad \exists \overline{v}, \overline{v}' : \Gamma(\overline{b}, \overline{b}', \overline{v}, \overline{v}') \tag{3.2}$$

The concrete transition relation $\mathcal{T}$ maps a concrete state $\overline{v}$ into a concrete next state $\overline{v}'$, and the abstract transition relation $\mathcal{B}$ maps a corresponding abstract state $\overline{b}$ into a corresponding abstract next state $\overline{b}'$. The abstraction function $\overline{\pi}$ maps the concrete states into abstract states. Put together, we get the classical abstraction connection:



Every satisfying assignment to (3.1) represents a concrete transition and its corresponding abstract transition. We aim at obtaining all possible satisfying assignments to the abstract variables $\overline{b}$ and $\overline{b}'$, that is, the set

$$\{(\overline{b},\overline{b}') \mid \mathcal{B}(\overline{b},\overline{b}')\} \tag{3.3}$$

This set is obtained by modifying the SAT solver Chaff as follows: every time a satisfying assignment is found, the tool records the values of the literals corresponding to the abstract variables $\overline{b}$ and $\overline{b}'$. Then the tool adds a blocking clause in terms of these literals that eliminates all satisfying assignments where these variables have the newly found values. The literals in the blocking clauses all have a decision level, since the assignment is complete. The solver then backtracks to the highest of these decision levels and continues its search for further, different satisfying assignments. Thus, the SAT solver is used to enumerate the set (3.3). This technique is commonly used in other areas; for example, by McMillan [McMillan 02] and by Gupta, Yang, and Ashar [Gupta 00]. Section 3.4 contains more details on how to efficiently obtain the set of satisfying assignments.

As an example, consider the following basic block:

```
d=e;
e++;
```

where `d` and `e` are integer variables. Suppose that predicates $\pi_1 = d\&1$ and $\pi_2 = e\&1$ are given. The binary operator & is the bit-wise conjunction operator—that is, π_1 holds if and only if `d` is odd, and π_2 holds if and only if `e` is odd. The basic block is translated into the following equation system, which represents the transition relation:

$$d_1 = e_0 \wedge e_1 = e_0 + 1 \tag{3.4}$$

By adding the required constraints according to Equation (3.2), we get

$$\begin{aligned} b_1 = d_0\&1 \wedge b_2 = e_0\&1 \wedge \\ d_1 = e_0 \wedge e_1 = e_0 + 1 \wedge \\ b_1' = d_1\&1 \wedge b_2' = e_1\&1 \end{aligned} \tag{3.5}$$

The satisfying assignments for this equation over the variables b_1, b'_1, b_2, and b'_2 are

b_1	b_2	b'_1	b'_2
0	0	0	1
0	1	1	0
1	0	0	1
1	1	1	0

In particular, the abstract Boolean program will never make a transition into a state that is contradictory in the sense that both `d` and `e` (which is equal to `d + 1`) are odd. This is unavoidable if a next state function is computed separately for each Boolean variable b_i, as many existing tools do.

Consider the basic block above with the predicates $\pi_1 = e \geq 0$ and $\pi_2 = e \leq 100$ and suppose that `x` has 32 bits. The equation for the abstract transition relation $\mathcal{B}$ is

$$\begin{aligned} b_1 = e_0 \geq 0 \wedge b_2 = e_0 \leq 100 \wedge \\ d_1 = e_0 \wedge e_1 = e_0 + 1 \wedge \\ b'_1 = e_1 \geq 0 \wedge b'_2 = e_1 \leq 100 \end{aligned} \tag{3.6}$$

The satisfying assignments for this equation over the variables b_1, b'_1, b_2, and b'_2 are

b_1	b_2	b'_1	b'_2
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	1	0
1	1	1	1

Note that incrementing a positive number is not guaranteed to yield another positive number because of the finite range. There is a transition from a state with $b_1 = 1$ to a state with $b'_1 = 0$.

3.3.2 The Abstract Transition Relation for Control-Flow Statements

Besides basic blocks, the concrete program also contains control-flow statements such as `if` and `while`. These statements take a condition as an argument and affect only the control flow (the program counter). We preprocess the program to remove all side effects from conditions. Since control-flow statements do not change the values of variables, we do not require an equation system to represent them.

Assume that we are abstracting a specific program counter location l that evaluates a condition c and moves the program counter to location l_T if c holds and l_F otherwise. Our

goal is to generate two sets of abstract transitions: (1) a set of transitions that assigns l_T to the program counter and (2) a set that assigns l_F. All the transitions will leave the abstract variables $\overline{b}$ unchanged.

To abstract c, we first traverse its syntactic structure to see whether there are any subexpressions that are also predicates in $\mathcal{P}$. We replace any occurrence of a predicate π_i in c with the corresponding Boolean variable b_i. Let c_1 be the condition that results from applying this transformation. If c_1 references only Boolean variables, we are done—this condition can be used in the abstract program. We then generate an abstract statement that assigns the program counter with l_T if $c_1(\overline{b})$ holds and l_F otherwise.

If, however, c_1 still refers to some concrete variables $\overline{v}$, we use the SAT enumeration engine to produce the set of abstract transitions that corresponds to the evaluation of c.

The condition $c(\overline{v})$ holds in an abstract state $\overline{b}$ if and only if there is a concrete state $\overline{v}$ such that the condition holds in $\overline{v}$ and $\overline{v}$ is mapped to $\overline{b}$. To create the abstract transition relation at this control location, we need to produce the set pos_c of abstract states from which there is a transition that assigns the program counter with l_T:

$$pos_c = \{\overline{b} \mid \exists \overline{v} : c(\overline{v}) \wedge \overline{\pi}(\overline{v}) = \overline{b}\} \tag{3.7}$$

The dual set neg_c of abstract states from which there is a transition that assigns the program counter with l_F is *not* the negation of pos_c. This act is true because a single abstract state can correspond to both concrete states that satisfy c and those that do not. We are therefore required to generate the set neg_c according to its definition:

$$neg_c = \{\overline{b} \mid \exists \overline{v} : \neg c(\overline{v}) \wedge \overline{\pi}(\overline{v}) = \overline{b}\} \tag{3.8}$$

Both of these sets are computed using the SAT enumeration engine.

In practice, we are rarely required to use the SAT enumeration engine for control-flow statements. The conditions of `if` statements and `while` loops are often chosen as Boolean predicates. Furthermore, most refinement algorithms will add these conditions whenever they encounter a spurious counterexample that passes through this statement.

3.4 The Implementation

3.4.1 Minimizing the Number of Quantified Variables

The size of the set (3.3) described in the previous section can be exponential in the number of predicates. However, in practice, a basic block usually mentions a very small subset of all program variables. Thus, most Boolean program variables are usually unchanged in the abstract version of the basic block. In particular, if a predicate uses only variables that are not assigned to, the truth value of the predicate is guaranteed not to change. We call the remaining predicates (the predicates that use variables that get assigned into) the *output predicates*. Formally, these are the predicates π_i, such that $\pi_i(v) \neq \pi_i(v')$.

Furthermore, we try to detect which predicates can actually influence the next abstract values of the output predicates. We do this by obtaining the set of variables that are used in

the assignments to variables that are mentioned in output predicates. We call these predicates the *input predicates*.

Example. As an example, consider the predicates $\pi_1 = \texttt{i} > \texttt{10}$ and $\pi_2 = \texttt{j} > \texttt{10}$. Let the basic block consist only of the statement

```
i=j;
```

In this case, π_1 is the only output predicate (as `j` is not modified), and π_2 is the only input predicate (as `i` is not mentioned in the right-hand side).

As an optimization, we only obtain the projection of the set (3.3) on the input and output predicates, where $\overline{b}$ is restricted to contain only input predicates and $\overline{b}'$ is restricted to contain only output predicates.

3.4.2 Obtaining the Set of Satisfying Assignments

The problem of obtaining the set of satisfying assignments to a formula restricted to a given subset of the variables corresponds to a quantification problem. Let S denote the subset of variables. We obtain the set by enumeration on the variables in S using a SAT solver. Plaisted [Plaisted 00, Plaisted 03] suggested this method for solving quantified formulae. In the work of Lahiri, Bryant, and Cook [Lahiri 03], our implementation algorithm was applied to predicate abstraction for hardware and software systems. It outperformed methods based on using Binary Decision Diagrams (BDDs) on all software examples. These results were obtained using arithmetic on integers however, not on bit-vectors.

The basic algorithm works as follows: when the SAT solver finds a satisfying assignment, it generates a blocking clause in terms of the variables in S. This clause is added to the clause database and prohibits any further satisfying assignment with the same values for the variables in S. After adding the clause to the CNF, the algorithm performs backtracking to the highest decision level of any of the variables in the blocking clause and continues the search for more satisfying assignments. Eventually, the additional constraints will make the problem unsatisfiable, and the algorithm will terminate. The blocking clauses added by the algorithm are a DNF representation of the desired set.

Each DNF clause represents a hypercube and is contained in the set of solutions. The basic algorithm can be improved by heuristics that try to enlarge the cube represented by each clause. McMillan uses conflict graph analysis in order to enlarge the cube [McMillan 02]. In Gupta, Yang, and Ashar's work [Gupta 00], BDDs are used for the enlargement. However, these techniques are beyond the scope of this report.

3.4.3 Using SMV to Check the Abstract Program

In our experiments, we used the Carnegie Mellon University version of SMV to verify the abstract program [K. McMillan 92, K. McMillan 93]. The advantage of using SMV is that the hypercubes representing the abstract transition relation can be passed to SMV directly by means of the `TRANS` statement. However, any other unbounded model checker is applicable as well, including SAT-based model checkers.

The control flow of the abstract program (which matches the control flow of the concrete program) is realized by adding a program counter variable. Each control-flow location corresponds to a set of hypercubes.

For the second example in Section 3.3, we obtain four cubes representing the six satisfying assignments:

$$\begin{array}{cccccccc} & \neg b_1 & \wedge & b_2 & \wedge & \neg b_1' & \wedge & b_2' \\ \vee & b_1 & \wedge & \neg b_2 & \wedge & \neg b_1' & \wedge & b_2' \\ \vee & b_1 & & & \wedge & b_1' & \wedge & \neg b_2' \\ \vee & & & b_2 & \wedge & b_1' & \wedge & b_2' \end{array}$$

Assuming that the program counter (PC) of this statement is `x`, the above cubes correspond to the following `TRANS` statement:

```
TRANS PC=x -> (!b1 &  b2 & !next(b1) &  next(b2))
            | ( b1 & !b2 & !next(b1) &  next(b2))
            | ( b1       &  next(b1) & !next(b2))
            | (       b2 &  next(b1) &  next(b2))
```

3.4.4 Simulating the Abstract Counterexample

If the model checker detects that the property does not hold on the abstract program, it generates a counterexample trace. This trace is then simulated on the concrete program in order to determine whether the counterexample is spurious. Most existing tools use a theorem prover such as Simplify for this task [Detlefs 03].

The disadvantages of using a general-purpose theorem prover for the simulation of the counterexample are similar to the disadvantages that arise during the computation of the abstract transition relation: The set of operators is limited, and the theorem prover may misjudge a counterexample to be real due to the lack of overflow detection.

The methodology that is used to obtain the concrete transition relation is also applicable to simulating the counterexample. Following the control flow in the abstract trace, we concatenate the corresponding basic blocks of the concrete program and apply the symbolic simulation technique described earlier.

Then, incrementally add the constraints that the control flow in the abstract trace impose, that is, the concretized versions of the control-flow conditions. After adding a new control-flow condition as a constraint, we check the satisfiability of the equation using SAT. If the equation is satisfiable, the abstract trace can be simulated so far. If it is unsatisfiable, the abstract trace cannot be simulated and is therefore spurious.

If all control-flow conditions found in the abstract trace are added and the equation is still satisfiable, the abstract trace can be simulated on the concrete program, and thus, a bug has been found. The tool then prints out the concrete trace. The values of the concrete variables can be obtained directly from the satisfying assignment.

In comparison to the concrete program, the control-flow conditions are small. Thus, only a few clauses and variables are added to the CNF in each step. Therefore, we use an

incremental SAT solver [Moskewicz 01] in order to preserve the information learned by the solver between the satisfiability checks.

3.4.5 Verifying Properties of the Program

The setup described so far can be used to check the reachability of code locations, as done by other tools such as SLAM, BLAST, or MAGIC. In addition to that, we check several safety properties such as array bounds and user-defined assertions.

The ANSI-C standard stipulates that, at any point in the program, one can insert an `assert` statement that specifies a Boolean condition. For example, the program

```
x = y;
y = y + 1;
assert(y > x);
```

asserts that after the two assignments, `y` will be greater than `x`. This assertion fails if incrementing `y` results in an overflow. Assertions are placed in the program as a specification of correctness. In order to verify the program, we must determine that the condition in the assertion is true in all possible executions.

When creating the abstract program, we translate every `assert(C)` statement, where `C` is a Boolean condition, by abstracting the condition `C`. This translation is done using the same method that we use for the conditions of "`if`" and "`while`" statements, as described in Section 3.3.2.

In addition to user-specified assertions, we verify several basic correctness properties of the program:

- Whenever a basic block contains a dereference $*p$ of a pointer variable p, we check that the pointer cannot be pointing to an illegal address. Let $\theta(p, \mathcal{P})$ be the set of variables for which the predicates can imply that p is pointing to (Definition 4). We then check that the following

  $$\bigvee_{v \in \theta(p,\mathcal{P})} (p = \&v)$$

  is valid by abstracting the expression as described in Section 3.3.2.

- Whenever a basic block contains a reference to an element of an array, we make sure that the array boundaries are not violated. If the expression $a[i]$ appears in the basic block (where i may be any integer expression) and the array a is of length n, we check the following for validity:

  $$(i < n) \wedge (i \geq 0)$$

- Whenever the basic block contains an expression that performs division (i.e., an expression of the form x/y [where y can be any numeric expression]), we make sure that the divisor is not zero.

3.5 Experimental Results

We applied the SAT-based abstraction approach to the verification of several C programs, four of which are discussed below.

3.5.1 Secure Hash Algorithm (SHA)

We used a program taken from the Digital Signature Standard (DSS). Under the DSS, communication among remote parties is enabled using digital signatures. The digital signature is computed using two inputs: 1) a delivery message of the communication instance and 2) a private key of a public/private key pair. We verified the C implementation of the DSS Secure Hash Algorithm (SHA) [NIST 95].

The SHA computes a part of the DSS digital signature called the message digest. The hashing algorithm computes the message digest by generating a 160-bit representation of the delivery message. The hashing procedure is designed to assure that the digest is statistically unique. The implementation makes extensive use of bit-wise operators and division.

The code contains calls to `abort()` in places where an unexpected condition happens, such as an arithmetic error. These calls can be considered an implicit property. We replace these calls by `assert(0)`—that is, we prove that these program locations are not reachable. The reachability of one of these locations depends on the result of a division: the code divides a 32-bit variable `t` by 20 and then checks that the result is between 0 and 3 using a `switch` statement. As discussed in Figure 5, if the result is any other value (default case), `abort()` is called. The property holds as the range of `t` is limited appropriately.

Given one predicate for each of the four possible `switch` cases, our tool generates an abstract transition relation that is consistent (at most one of the four, mutually exclusive predicates holds) and strong enough to show the property (at least one of the four predicates holds). The overall runtime (including preparation, one refinement iteration, and the SMV runtime) is 24 seconds on a 2 GHZ machine, most of which is spent within the SAT solver. The predicate abstraction tools described in the related work generate an abstraction that lacks at least the last property—that is, the result of the division is between zero and three.

3.5.2 ASN1 Data Structures in OpenSSL

OpenSSL comes with an implementation of ASN1 data structures for managing certificates [Freier 96]. Using an if-then-else construct, individual bits of a variable `j` are tested. The variable has type `signed int`. Previously, the integer was assigned the value of an unsigned char array member. The array member is known to be nonzero. Therefore, the code, as shown in Figure 6, assumes that one of the first eight bits must be set. Proving the assertion requires a bit-vector decision procedure; the assertion is not part of the original code.

Within one refinement iteration, the following predicates are obtained: one predicate that holds if the array member is nonzero and one predicate for each of the branching guards. Using these predicates, our tool generates an abstract transition relation that enforces that exactly one of these predicates is true, allowing the model checker to show that the assertion is not reachable.

```
switch ( t / 20 )
  {
   case 0:
      TEMP2 = ( (B AND C) OR (~B AND D) );
      TEMP3 = ( K_1 );
      break;

   case 1:
      TEMP2 = ( (B XOR C XOR D) );
      TEMP3 = ( K_2 );
      break;

   case 2:
      TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
      TEMP3 = ( K_3 );
      break;

   case 3:
      TEMP2 = ( B XOR C XOR D );
      TEMP3 = ( K_4 );
      break;

   default:
      assert(0);
  }
```

Figure 5: Excerpt from an SHA Implementation

```
int ret,j,bits,len;
. . .

  j=a->data[len-1];
  if      (j & 0x01) bits=0;
  else if (j & 0x02) bits=1;
  else if (j & 0x04) bits=2;
  else if (j & 0x08) bits=3;
  else if (j & 0x10) bits=4;
  else if (j & 0x20) bits=5;
  else if (j & 0x40) bits=6;
  else if (j & 0x80) bits=7;
  else { bits=0; assert(0); } /* should not happen */
```

Figure 6: Excerpt from an Implementation of ASN1 Data Structures from OpenSSL

3.5.3 MD2 Message-Digest Algorithm

Similar to the SHA algorithm, the MD2 message-digest algorithms computes a hash of a given message. RFC 1319 gives a reference implementation in ANSI-C [Koliski 92]. A part of it is shown in Figure 7. The algorithm makes extensive use of a permutation that is given as an array. In the first part, the result of the previous iteration is used as an array index for the next iteration. The second part uses the bit-wise xor of the result of the previous iteration and a part of the message as an array index.

We verify that these lookups do not violate the bounds of the `PI_SUBST` array. As the variable `t` is of an unsigned integer type, only the upper array bound can be violated—that is, the predicate `t<256` must hold in the first part, and the predicate `(block[i]^t)<256` must hold in the second part of the algorithm. For each of the four code locations `t` is modified in, the SAT solver easily discovers that these predicates indeed are true in the next state.

3.5.4 Pointer Arithmetic in JPEG Decoder

For efficiency reasons, many programs use pointer arithmetic instead of array index expressions within loops. As an example, consider the code in Figure 8 from a JPEG decoder [VisicronCorporation 03]. It performs a discrete cosine transformation using a loop that iterates through an array of 64 elements. Each loop iteration processes one row, which corresponds to `DCTSIZE=8` array elements. Thus, iteration number `ctr` accesses the elements `data[8*(7-ctr)]` to `data[8*(7-ctr)+7]`. To avoid this computation for each array access, the code uses a pointer that points to `data[8*(7-ctr)]`. This pointer is then used to access the individual elements.

To prove that the pointer access happens within the array bounds, we use the predicates `dataptr==&data[8*(7-ctr)]`, `ctr>=0`, and `ctr<DCTSIZE`.

3.6 Related Work

Abstraction techniques are often based on the abstract interpretation work of Cousot and Cousot [Cousot 77] and require the user to give an abstraction function relating concrete datatypes to abstract datatypes. Earlier applications of the predicate abstraction type of the abstract interpretation approach [Graf 97, Bensalem 98, Colón 98] require the user to identify the set of predicates that influence the verification property and utilize general-purpose theorem proving to compute the abstract program. The user-driven discovery of relevant predicates makes these methods less effective for large programs.

Recently, various decision procedures have been proposed to compute the set of predicates for abstraction. The most common approach is to use error traces [Clarke 00, Ball 01a] to guide the predicate discovery. In the work of Clarke and colleages [Clarke 00], the algorithm is based on BDD representations of the program. The BDD-based approach is not efficient when it comes to analysis of large programs, where transition relation BDDs are commonly too large for efficient manipulation. The algorithm presented in the work of Ball and colleages [Ball 01a] uses an explicit state representation, but it is restricted to safety properties.

```
static unsigned char PI_SUBST[256] =
   ...
;

static void MD2Transform (state, checksum, block)
unsigned char state[16];
unsigned char checksum[16];
unsigned char block[16];
{
  unsigned int i, j, t;
  unsigned char x[48];

  /* Form encryption block from state, block, state ^ block.
   */
  MD2_memcpy ((POINTER)x, (POINTER)state, 16);
  MD2_memcpy ((POINTER)x+16, (POINTER)block, 16);
  for (i = 0; i < 16; i++)
    x[i+32] = state[i] ^ block[i];

  /* Encrypt block (18 rounds).
   */
  t = 0;
  for (i = 0; i < 18; i++) {
    for (j = 0; j < 48; j++)
      t = x[j] ^= PI_SUBST[t];   /* t must be <= 255 */
    t = (t + i) & 0xff;
  }

  /* Save new state */
  MD2_memcpy ((POINTER)state, (POINTER)x, 16);

  /* Update checksum.
   */
  t = checksum[15];
  for (i = 0; i < 16; i++)
    t = checksum[i] ^= PI_SUBST[block[i] ^ t]; /* t must be <= 255 */

  /* Zeroize sensitive information.
   */
  MD2_memset ((POINTER)x, 0, sizeof (x));
}
```

Figure 7: Excerpt from the Reference Implementation of the MD2 Algorithm

```
jpeg_fdct_ifast (DCTELEM * data)
{
  ...
  DCTELEM *dataptr;
  int ctr;
  ...

  /* Pass 1: process rows. */

  dataptr = data;
  for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[0] + dataptr[7];
    ...

    dataptr += DCTSIZE; /* advance pointer to next row */
  }
  ...
```

Figure 8: Excerpt from JPEG decoder

The abstraction refinement loop was first introduced by Kurshan [Kurshan 95]. Kurshan's localization reduction technique produces an initial abstraction of the program by "freeing away" program variables that do not affect the verification property. The values of "free" variables are defined nondeterministically, which results in an overapproximation of the program behaviors. The unrealistic behaviors are eliminated from the program by gradually refining the "free" variables back to their original values.

Clarke and colleages [Clarke 00] extended the work of Kurshan by defining a procedure for systematic abstraction refinement. Spurious error traces are used by the refinement decision procedure to ensure that the new abstraction will not allow this counterexample.

Predicate abstraction of ANSI-C programs in combination with CEGAR has become a widely applied technique. It was introduced by Ball and Rajamani [Ball 00] and promoted by the success of the SLAM project. The goal of SLAM was to verify that Windows device drivers obey application program interface (API) conventions. SLAM models program variables using unbounded integer numbers and does not take overflow or bit-wise operators into account. The abstraction of the program is computed using a theorem prover such as Simplify [Detlefs 03]. The property checked mainly depends on the control flow, and thus, this treatment is sufficient. However, there are C programs that make extensive use of bit-wise operators; for example, programs that represent a circuit model. For these programs, we expect that the limited range of the variables will be crucial. BOOP [Weissenbacher 03] is a reimplementation of SLAM. BLAST is another software model checker for C programs that uses the CEGAR approach to construct an abstract program. The abstraction is constructed "on the fly" and only to the required precision.

The NASA Ames model checker, JavaPathFinder [Brat 00], developed for verifying Java programs, was also reported to use heuristics for automated predicate abstraction and

refinement. In this tool, predicate abstraction procedures are extended with some informal abstraction arguments that allow predicate abstraction to be used within the class instance of object-oriented languages. The MAGIC tool, applies automatic compositional reasoning on programs with functions.

Recently, there has been some work reported on the application of SAT solvers in the process of constructing and refining predicate abstractions. Previously, the application of SAT solvers during computation of predicate abstraction was conducted only in the context of hardware verification in the work of Clarke, Talupur, and Wang [Clarke 03]. The focus of that work, indeed, is the refinement of the initial approximate abstraction and not the construction of the abstraction itself. The approximate abstract model is constructed by excluding certain implications from consideration. In contrast, we use a SAT solver to construct the exact abstract transition relation according to the provided set of predicates, rather than an approximation of it.

Strichman and colleages [Chaki 03b] use a SAT engine for identifying (or approximating) the minimal set of predicates needed to eliminate a set of spurious counterexamples during refinement of abstract C programs. The predicate-minimizing algorithm is implemented in the MAGIC tool, which uses a theorem prover to compute predicate abstraction.

The feature that distinguishes the ComFoRT SAT-based approach from the existing tools and approaches is the tight integration of a SAT solver into the abstraction. The SAT-based abstraction approach allows precise encodings of the semantics of the ANSI-C language, including pointer-arithmetic and bit-vector overflow. In contrast to that, all the tools mentioned above rely on external theorem provers to reason about the programming language constructs. Initially, SLAM, BLAST, and MAGIC used the theorem prover Simplify, which supports linear arithmetic on real numbers only. The remaining operators are approximated by means of uninterpreted functions. The SLAM project replaced Simplify with Zapato [Ball 04], which provides better performance but no support for bit-vectors.

The use of propositional logic and a SAT solver to reason about ANSI-C language constructs is already found in a bounded model checker for ANSI-C programs, CBMC [Clarke 04d]. A prototype of SLAM has been integrated with parts of CBMC to reason about bit-vector constructs [Cook 05]. This version has found a previously unknown bug in Windows.

Lahiri, Bryant, and Cook [Lahiri 03] use the SAT-based quantification engine fron the work of Clarke and colleages [Clarke 04e] in order to compute abstractions of C programs. However, the algorithm uses unbounded integer semantics for the program variables and does not support bit-vector operators.

4 Conclusion

This report presented one of the ComFoRT model checking techniques, namely a SAT-based predicate abstraction approach [Clarke 04e]. This method replaces the use of theorem provers with the use of a SAT solver. The report demonstrated that SAT-based predicate abstraction outperforms the approaches that use theorem provers, since enumeration on a single SAT instance can substitute for a potentially exponential number of theorem prover calls. The advantages are particularly pronounced when the number of abstract transitions is significantly smaller than the number of possibilities that need to be checked. Furthermore, since modern SAT solvers allow for the evaluation of a large number of possible assignments to the abstract program variables, the application of a SAT engine results in a more precise transition relation of the abstract program compared to the abstraction produced by using theorem provers. The SAT-based approach results in eliminating some unrealistic behaviors of the abstract program that otherwise would be introduced during the overapproximations of the abstract transition relation computed using a theorem prover.

Model checking a more precise abstract program, therefore, exhibits a smaller number of redundant spurious counterexamples. As a result, fewer CEGAR loop iterations are required until the verification property is confirmed or refuted. The latter fact is of high value to practical software verification, since the validation of counterexamples and predicate refinement (Steps 3 and 4 of the CEGAR loop) are computationally expensive. Our approach, therefore, simplifies (if not enables) the application of model checking to the verification of large-scale programs by eliminating analysis and refinement of redundant counterexamples.

Another advantage of the SAT-based abstraction technique is that most ANSI-C constructs can be handled during the program abstraction. The ability to handle low-level programming constructs of ANSI-C differs from other model checking approaches that operate on only a small subset of the C language. The SAT-based approach enables model checking of realistic programs by supporting the more complex features of C, such as multiplication/division, pointers, bit-wise operations, type conversion, and shift operators.

A notable advantage of the SAT-based abstraction technique is that it can be reused within the CEGAR loop without any changes to the error trace simulation and predicate refinement used in the loop.

The SAT-based abstraction technique has been implemented in a SATABS tool [Clarke 05] that can be invoked within the ComFoRT framework. In the future, we plan to couple the SATABS tool with the current model checking engine of ComFoRT to enable closer integration.

References

[Ball 00] Ball, T. & Rajamani, S. *Boolean Programs: A Model and Process for Software Analysis*. Redmond, WA: Microsoft Research, February 2000.

[Ball 01a] Ball, T.; Majumdar, R.; Millstein, T.; & Rajamani, S. "Automatic Predicate Abstraction of C Programs", 203–213. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY: Association for Computing Machinery, 2001.

[Ball 01b] Ball, T. & Rajamani, S. K. "Automatically Validating Temporal Safety Properties of Interfaces", 103–122. *Proceedings of the* 8^{th} *International SPIN Workshop on Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science (LNCS)*. Toronto, Canada, May 19-20, 2001. New York, NY: Springer, 2001.

[Ball 04] Ball, T.; Cook, B.; Lahiri, S. K.; & Zhang, L. "Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement", 457–461. *Proceedings of the* 16^{th} *International Conference on Computer-Aided Verification (CAV)*. Boston, MA, July 13-17, 2004. New York, NY: Springer, 2004.

[Bensalem 98] Bensalem, S.; Lakhnech, Y.; & Owre, S. "Computing Abstractions of Infinite State Systems Compositionally and Automatically", 319–331. *Proceedings of the* 10^{th} *International Conference on Computer-Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science (LNCS)*. Vancouver, BC, Canada, June 28-July 2. Berlin, Germany: Springer, 1998.

[Brat 00] Brat, G.; Havelund, K.; Park, S.; & Visser, W. "Java PathFinder - a Second Generation of a Java Model Checker", 130–135. *Proceedings of Post-CAV Workshop on Advances in Verification*. Chicago, Illinois, July 20, 2000. http://citeseer.ist.psu.edu/cache/papers/cs/15878/http:zSzzSzwww.riacs.eduzSzresearchzSzdetailzSzasezSzwave00.pdf/visser00java.pdf (2000).

[Chaki 03a] Chaki, S.; Clarke, E.; Groce, A.; Jha, S.; & Veith, H. "Modular Verification of Software Components in C", 385–395. *Proceedings of the* 25^{th} *International Conference on Software Engineering (ICSE)*. Portland, OR, May

3-10, 2003. Los Alamitos, CA: Institute of Electrical and Electronics Engineers, 2003.

[Chaki 03b] Chaki, S.; Clarke, E.; Groce, A.; & Strichman, O. "Predicate Abstraction with Minimum Predicates", 19–34. *Proceedings of the* 12^{th} *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *Lecture Notes in Computer Science (LNCS)*. L'Aquila, Italy, October 21-24, 2003. New York, NY: Springer, 2003.

[Chaki 04] Chaki, S.; Sharygina, N.; & Sinha, N. "Verification of Evolving Software". *Proceedings of the Third International Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*. Newport Beach, CA, October 31-November 1, 2004, 2004. http://www.cs.iastate.edu/~leavens/SAVCBS/2004/papers/Chaki-Sharygina-Sinha.pdf.

[Clarke 82] Clarke, E. M. & Emerson, E. A. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic". *Logics of Programs Workshop*, volume 131 of *Lecture Notes in Computer Science (LNCS)*. New York, NY, May 1981. Berlin, Germany: Springer, 1982.

[Clarke 86] Clarke, E. M.; Emerson, E. A.; & Sistla, A. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems 8*, 2 (April 1986): 244–263.

[Clarke 92] Clarke, E.; Grumberg, O.; & Long, D. "Model Checking and Abstraction", 343–354. *Proceedings of the* 19^{th} *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Albuguerque, NM, January 19-22, 1992. New York, NY: Association for Computing Machinery, 1992.

[Clarke 99] Clarke, E.; Grumberg, O.; & Peled, D. *Model Checking*. Cambridge, MA: MIT Press, December 1999.

[Clarke 00] Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; & Veith, H. "Counterexample-Guided Abstraction Refinement", 154–169. *Proceedings of the* 12^{th} *International Conference on Computer Aided Verification (CAV 2000)*. Chicago, IL, July 15-19, 2000. Berlin, Germany: Springer, 2000.

[Clarke 03] Clarke, E.; Talupur, M.; & Wang, D. "SAT-Based Predicate Abstraction for Hardware Verification", 78–92.

Theory and Applications of Satisfiability Testing: Sixth International Conference (SAT 2003). Santa Margherita Ligure, Italy, May 5-8, 2003. New York, NY: Springer, 2003.

[Clarke 04a] Clarke, E.; Chaki, S.; Groce, A.; Ouaknine, J.; Strichman, O.; & Yorav, K. "Efficient Verification of Sequential and Concurrent C Programs". *Formal Methods in System Design 25*, 2-3 (September/November 2004): 129–166.

[Clarke 04b] Clarke, E. M.; Chaki, S.; Grumberg, O.; Ouaknine, J.; Sharygina, N.; Touili, T.; & Veith, H. *An Expressive Verification Framework for State/Event Systems* (CMU-CS-04-145). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, 2004.

[Clarke 04c] Clarke, E. M.; Chaki, S.; Ouaknine, J.; Sharygina, N.; & Sinha, N. "State/Event-Based Software Model Checking", 128–147. *Proceedings of the* 4^{th} *International Conference on Integrated Formal Methods (IFM 04)*. Canterbury, UK, April 4-7, 2004. Berlin, Germany: Springer, 2004.

[Clarke 04d] Clarke, E.; Kroening, D.; & Lerda, F. "A Tool for Checking ANSI-C Programs", 168–176. *Tools and Algorithms for the Construction and Analysis of Systems:* 10^{th} *International Conference (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science (LNCS)*. Barcelona, Spain, March 29-April 2, 2004. New York, NY: Springer, 2004.

[Clarke 04e] Clarke, E.; Kroening, D.; Sharygina, N.; & Yorav, K. "Predicate Abstraction of ANSI–C Programs using SAT". *Formal Methods in System Design (FMSD) 25*, 2–3 (September–November 2004): 105–127.

[Clarke 05] Clarke, E.; Kroening, O. D.; Sharygina, N.; & Yorav, K. "SATABS: SAT-Based Predicate Abstraction for ANSI-C", 570–574. *Tools and Algorithms for the Construction and Analysis of Systems:* 11^{th} *International Conference (TACAS 2005)*. Edinburgh, Scotland, UK, April 4-8, 2005. New York, NY: Springer, April 2005.

[Colón 98] Colón, M. & Uribe, T. "Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures", 293–304. *Computer-Aided Verification:* 10^{th} *International Conference (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science (LNCS)*. Vancouver, BC, Canada, June 28-July 2, 1998. Berlin, Germany: Springer, 1998.

[Cook 05] Cook, B.; Kroening, D.; & Sharygina, N. "Cogent: Accurate Theorem Proving for Program Analysis". *Computer-Aided Verification:* 17^{th} *International Conference (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science (LNCS)*. Edinburgh, Scotland, UK, July 6-10, 2005. New York, NY: Springer, 2005.

[Cousot 77] Cousot, P. & Cousot, R. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", 238–252. *Proceedings of the* 4^{th} *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. Los Angeles, CA, January 17-19, 1977. New York, NY: Association for Computing Machinery, 1977.

[Currie 00] Currie, D. W.; Hu, A. J.; Rajan, S.; & Fujita, M. "Automatic Formal Verification of DSP Software", 130–135. *Proceedings 2000:* 37^{th} *Design Automation Conference*. Los Angeles, CA, June 5-9, 2000. New York, NY: Association for Computing Machinery, 2000.

[Das 01] Das, S. & Dill, D. "Successive Approximation of Abstract Transition Relations". *Proceedings of the* 16^{th} *Annual IEEE Symposium on Logic in Computer Science (LICS)*. Boston, MA, June 16-19, 2001. Los Alamitos, CA: IEEE Computer Society, 2001.

[Detlefs 03] Detlefs, D.; Nelson, G.; & Saxe, J. B. *Simplify: A Theorem Prover for Program Checking* (HPL-2003-148). Palo Alto, CA: HP Labs, July 2003.

[Freier 96] Freier, A.; Karton, P.; & Kocher, P. "The SSL Protocol, Version 3.0", 1996. http://wp.netscape.com/eng/ssl3/ssl-toc.html (March 1996).

[Graf 97] Graf, S. & Saïdi, H. "Construction of Abstract State Graphs with PVS", 72–83. *Proceedings of the Ninth International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science (LNCS)*. Haifa, Israel, June 22-25, 1997. New York, NY: Springer, 1997.

[Gries 80] Gries, D. & Levin, G. "Assignment and Procedure Call Proof Rules". *ACM Transactions on Programming Languages and Systems (TOPLAS) 2*, 4 (October 1980): 564–579.

[Gupta 00] Gupta, A.; Yang, Z.; Ashar, P.; & Gupta, A. "SAT-Based Image Computation with Application in Reachability

Analysis", 354–372. *Formal Methods in Computer-Aided Design, Third International Conference (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science (LNCS)*. Austin, TX, November 1–3, 2000. New York, NY: Springer, 2000.

[Henzinger 02] Henzinger, T. A.; Jhala, R.; Majumdar, R.; & Sutre, G. "Lazy Abstraction", 58–70. *Proceedings of the* 29^{th} *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2002)*. Portland, OR, January 16-18, 2002. New York, NY: Association for Computing Machinery, 2002.

[Hissam 02] Hissam, S. & Ivers, J. *PECT Infrastructure: A Rough Sketch* (CMU-SEI-2002-TN-033, ADA413548). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports/02tn033.html.

[Ivers 04] Ivers, J. & Sharygina, N. *Overview of ComFoRT: A Model Checking Reasoning Framework* (CMU/SEI-2004-TN-018). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports/04tn018.html.

[K. McMillan 92] K. McMillan. *Symbolic Model Checking: An Approach to the State Space Explosion Problem (CMU-CS-92-131)*. Pittsburgh, PA, School of Computer Science: Carnegie Mellon University, 1992.

[K. McMillan 93] K. McMillan. *Symbolic Model Checking*. Boston, MA: Kluwer Academic, 1993.

[Koliski 92] Koliski, B. *RFC 1319 (RFC1319)*. 1992. http://www.faqs.org/rfcs/rfc1319.html (1992).

[Kroening 03a] Kroening, D.; Clarke, E.; & Yorav, K. "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking", 368–371. *Proceedings 2003:* 40^{th} *Design Automation Conference*. Anaheim, CA, June 2-6, 2003. New York, NY: Association for Computing Machinery, 2003.

[Kroening 03b] Kroening, D.; Clarke, E.; & Yorav, K. *Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking* (CMU-CS-03-126). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, 2003.

[Kurshan 95] Kurshan, R. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton, NJ: Princeton University Press, 1995.

[Lahiri 03] Lahiri, S. K.; Bryant, R. E.; & Cook, B. "A Symbolic Approach to Predicate Abstraction", 141–153. *Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science (LNCS)*. Boulder, CO, July 8-12, 2003. New York, NY: Springer, 2003.

[McMillan 02] McMillan, K. "Applying SAT Methods in Unbounded Symbolic Model Checking", 250–264. *Computer-Aided Verification:* 14^{th} *Conference (CAV 2002)*, volume 2402 of *Lecture Notes in Computer Science (LNCS)*. Copenhagen, Denmark, July 27-31, 2002. New York, NY: Springer, 2002.

[Moskewicz 01] Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; & Malik, S. "Chaff: Engineering an Efficient SAT Solver", 530–535. *Proceedings of the 38th Design Automation Conference (DAC'01)*. Las Vegas, NV, June 18-22, 2001. New York, NY: Association for Computing Machinery, 2001.

[NIST 95] National Institute of Standards and Technology, Computer Systems Laboratory. *Secure Hash Standard (FIPS 180-1)*, April 1995. Gaithersburg, MD: Computer Systems Laboratory, National Institute of Standards and Technology. http://www.nist.gov/itl/div897/pubs/fip180-1.htm.

[Plaisted 00] Plaisted, D. *Method for Design Verification of Hardware and Non-Hardware Systems* (U.S. Patent Number 6,131,078). October 10 2000. http://patents.cos.com/ (September 2005).

[Plaisted 03] Plaisted, D.; Biere, A.; & Zhu, Y. "A Satisfiability Tester for Quantified Boolean Formulae". *Discrete Applied Mathematics 130*, 2 (August 15 2003): 291–328.

[Pnueli 77] Pnueli, A. "The Temporal Logic of Programs". *18th Annual Symposium on Foundations of Computer Science*. Providence, RI, October 31-November 2, 1977. New York, NY: Institute of Electrical and Electronics Engineers, 1977.

[VisicronCorporation 03] VisicronCorporation. *JPEG*. 2003. http://datacompression.info/JPEG.shtml (2003).

[Wallnau 03a] Wallnau, K. *Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC)*

(CMU-SEI-2003-TR-009, ADA413574). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tr009.html.

[Wallnau 03b] Wallnau, K. & Ivers, J. *Snapshot of CCL: A Language for Predictable Assembly* (CMU-SEI-2003-TN-025, ADA418453). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tn025.html.

[Weissenbacher 03] Weissenbacher, G. "An Abstraction/Refinement Scheme for Model Checking C Programs". Master's diss., Graz University of Technology, Graz, Austria, 2003.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2005	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
SAT-Based Predicate Abstraction of Programs	FA8721-05-C-0003

6. AUTHOR(S)

Edmund Clarke, Daniel Kroening, Natasha Sharygina, Karen Yorav

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2005-TR-006

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2005-006

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Component Formal Reasoning Technology, ComFoRT, is a model-checking-based approach for analysis of component-based software designs. ComFoRT is designed to be used in a prediction-enabled component technology (PECT). A PECT provides a means to reliably predict the runtime qualities (e.g., performance and reliability) of assemblies of components from their certifiable properties (e.g., execution time and behavioral descriptions). ComFoRT uses an abstraction-based approach to cope with the complexity of analysis by reducing the size of the program models to be analyzed. This note presents technical details of a SAT-based predicate abstraction technique used in ComFoRT. The main advantage of the SAT-based method over conventional predicate abstraction techniques is that it does not require an exponential number of theorem prover calls for computing an abstract model. Additionally, the SAT-based approach computes a more precise and safe abstraction compared to existing predicate abstraction methods.

14. SUBJECT TERMS	15. NUMBER OF PAGES
formal verification, abstraction, software model checking	49

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL