



Software Engineering Institute

Certified Binaries for Software Components

Sagar Chaki
James Ivers
Peter Lee
Kurt Wallnau
Noam Zeilberger

September 2007

TECHNICAL REPORT
CMU/SEI-2007-TR-001
ESC-TR-2007-001

Predictable Assembly from Certifiable Components Initiative
Unlimited distribution subject to the copyright.



CarnegieMellon

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.

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.

Copyright 2007 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

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 Basic Concepts	3
3 Framework for Generating Certified Binaries	7
3.1 The Procedure	7
3.2 Trusted Computing Base	8
4 Certifying Model Checking	11
4.1 Iterative Predicate Abstraction and Refinement	11
4.2 Construction and Composition Language	12
4.3 Interpretation from CCL to C	14
4.4 Ranking Functions Generated by Copper	15
5 Certified Source-Code Generation	17
6 Certified Binary Generation	21
7 Related Work	23
8 Experimental Results	25
9 Conclusion	27
References	29

List of Figures

Figure 1:	Component in Analysis (Left) and C (Right) Forms	3
Figure 2:	Framework for Generating Certified Binaries	7
Figure 3:	Example of Software Verification Via Iterative Refinement	11
Figure 4:	CCL Specification for a Simple Component	13
Figure 5:	Graphical Depiction of the Component's State Machine	13
Figure 6:	Ranking Function in Terms of the C Program	16
Figure 7:	Ranking Function in Terms of the CCL Specification	16
Figure 8:	Invariants in Pin/C Code (Left) and Assembly Code (Right)	17
Figure 9:	Invariants and Code in Pin/C (Left) Assembly (Right)	21

Abstract

Proof-carrying code (PCC) and certifying model checking (CMC) are two established paradigms for obtaining objective confidence in the runtime behavior of a program. PCC enables the certification of low-level binary code against relatively simple (e.g., memory-safety) policies. In contrast, CMC provides a way to certify a richer class of temporal logic policies, but is typically restricted to high-level (e.g., source) code. In this report, an approach is presented to certify binary code against expressive policies, and thereby achieve the benefits of both PCC and CMC. This approach generates certified binaries from software specifications in an automated manner. The specification language uses a subset of UML statecharts to specify component behavior and is compiled to the Pin component technology. The overall approach thus demonstrates that formal certification technology is compatible with, and can indeed exploit, model-driven approaches to software development. Moreover, this approach allows the developer to trust the code that is produced without having to trust the tools that produced it. In this report details of this approach are presented and experimental results on a collection of benchmarks are described.

1 Introduction

In the current plug-and-play era, off-the-shelf programs are increasingly being made available as modules or components that are attached to an existing infrastructure. Often, such plug-ins are developed from high-level component specifications (such as UML statecharts) but are distributed in executable machine code, or binary form. In this report, we present a framework for generating trustworthy binaries from component specifications and for proving that binaries generated elsewhere satisfy specific policies.

Our approach builds on two existing paradigms for software certification: proof-carrying code and certifying model checking. Proof-carrying code (PCC) was originally proposed in a seminal paper by Necula and Lee [Necula 1996]. The essential idea underlying PCC is to construct a proof of the claim that a piece of machine code respects a desired policy. The proof is shipped along with the code to allow independent verification before the code is deployed, hence “proof-carrying.” In contrast, certifying model checking (CMC) [Namjoshi 2001] is an extension of model checking [Clarke 1982] that generates “proof certificates” for finite state models against a rich class of temporal logic policies. In recent years, CMC has been augmented with iterative abstraction-refinement to enable the certification of C source code [Henzinger 2002a, Chaki 2006].

PCC and CMC have complementary strengths and limitations. Specifically, while PCC operates directly on binaries, its applications to date have been restricted to relatively simple memory safety policies.¹ The progress of PCC has also been hindered by the need for manual intervention (e.g., to specify loop invariants). In contrast, CMC can certify programs against a richer class of temporal logic policies (which subsumes both safety and liveness) and is automated. However, CMC can certify only source code (for example “C”) or other forms of specification languages.

Finally, while PCC and CMC both require a small trusted computing base—usually consisting of a verification condition generator and a proof checker—they both tend to generate prohibitively large proofs. This can pose serious practical obstacles in using these techniques in resource-constrained environments. Unfortunately, the embedded software that can benefit most from the high confidence obtained with PCC (e.g., medical devices, on-board automotive systems, mobile phones) will likely be resource constrained.

In this context, our approach has the following salient features:

1. **expanded applicability:** We generate certified binaries directly from component specifications expressed in a subset of UML statecharts. The key technique involved is a process of translating “ranking functions,” along with the component itself, from one language to the next. Thus, our approach bridges the two domains of model-driven software development and formal software certification.
2. **rich policies:** As with CMC, we certify components against a rich class of temporal logic policies that subsume both safety and liveness policies. We use the state/event-based temporal logic called SE-LTL [Chaki 2004b] developed by the Predictable Assembly from Certifiable Components (PACC)² Initiative at the Carnegie Mellon^{®3} Software Engineering Institute (SEI).
3. **automation:** As with CMC, we employ iterative refinement in combination with predicate abstraction and model checking to generate appropriate invariants and

¹ Informally, a safety policy stipulates a condition that must never occur, while a liveness policy stipulates a condition that must eventually occur.

² For more information, go to <http://www.sei.cmu.edu/pacc/>.

³ [®]Carnegie Mellon is registered in the U.S. Patent and Trademark office by Carnegie Mellon University.

ranking functions required for certificate and proof construction in an automated manner.

4. **compact proofs:** We use state-of-the-art Boolean satisfiability (SAT) technology to generate extremely small proofs. Our results indicate that the use of SAT yields proofs of manageable size for realistic examples.

The rest of this report is organized as follows. In Section 2, we present the basic concepts relevant to our research; and in Section 3, we present an overview of our framework. We then focus on the CMC portion of our framework in Section 4, on the process of generating certified source code in Section 5, and on the process of generating certified binaries in Section 6. We survey related work in Section 7 and present some experimental results in Section 8. Finally, we conclude in Section 9.

2 Basic Concepts

In this section, we describe the basic concepts of components, policies, ranking functions, verification conditions, certificates, and resolutions that we use in the rest of this report.

Logical Foundation. We assume a denumerable set of variables *Var* and a set of expressions *Expr* constructed using *Var* and the standard C operators. We view every expression as a formula in quantifier-free first-order logic with C interpretations for operators and truth values (0 is false and anything else is true). Thus, we use the terms “expression” and “formula” synonymously and apply the standard concepts of validity, satisfiability, and so forth to both expressions and formulas.

Component. We deal with several forms of a component—its construction and composition language (CCL) form, C implementation form, analysis form, and its binary (assembly language) form. The syntax and semantics of CCL have been presented elsewhere [Wallnau 2003], and we use the PowerPC assembly language. Hence, we only describe the other two (analysis and C implementation) forms in more detail.

In its analysis form, a component is simply a control-flow graph (CFG) with a specific entry node. Each node of the component is labeled with either an assignment statement, a branch condition, or a procedure call. The outgoing edges from a branch node are labeled with **THEN** and **ELSE** to indicate flow of control. For any component *C*, we write *Stmt(C)* to denote the set of nodes of *C*, since each node corresponds to a component statement. Figure 1 shows a component on the left and its representation in C syntax on the right.

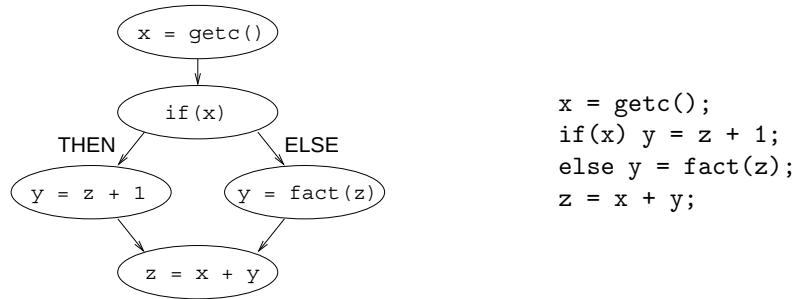


Figure 1: Component in Analysis (Left) and C (Right) Forms

The analysis form is a very abstract notion of component that enables us to transcend syntactic differences among different kinds of specification languages (CCL, C, or other) and therefore to talk about component certification in a very general way.

The C implementation is generated from CCL and contains both the logical behavior as specified by statecharts as well as the infrastructure imposed by the Pin component model [Hissam 2005]. However, we impose several strong restrictions on the C code itself. For instance, we disallow recursion so that the entire component is inlined into a single CFG. We also disallow internal concurrency. Variable scopes and return statements are not considered. All variables are assumed to be of integral type, and pointers and other complicated data types are disallowed.

While these are severe restrictions when viewed from the full generality of ANSI-C, they are not so severe when viewed from the more restrictive vantage of CCL specifications. In particular, a CCL specification for a component with a single reaction (the CCL unit of concurrency) obeys the above restrictions by definition. Even when a restriction is violated (e.g., CCL allows statically declared

fixed-size arrays), simple transformations (e.g., representing each array element by a separate variable) suffice to solve the problem. Since all C programs and binaries we consider are obtained via some form of semantics-preserving translation of CCL specifications, they obey our restrictions as well.

Policy. Policies are expressed in CCL specifications as SE-LTL formulas. Prior to verification, however, the policy is transformed into an equivalent Büchi automaton. Thus, for the purpose of this report, a policy φ is to be viewed simply as a Büchi automaton. The theoretical details behind the connection between SE-LTL and Büchi automata can be found elsewhere [Chaki 2004b] and are not crucial for grasping the main ideas presented here.

Ranking Function. Ranking functions are a technical device used to construct proofs of liveness, which require a notion of progress toward a specific objective *Obj*. The essential idea is to assign ranks—drawn from an ordered set R with no infinite decreasing chains—to system states. Informally, the rank of a state is a measure of its distance from *Obj*. Then, proving the liveness policy boils down to proving that the rank of the current system state decreases appropriately with every transition (i.e., the system makes progress toward *Obj*). Since there are no infinite decreasing chains in R , the system must eventually attain *Obj*. In our case, it suffices to further restrict R to be a finite set of integers with the usual ordering.

Definition 1 (Ranking Function) *Given a component C , a policy φ , and a finite set of integral ranks R , a ranking function RF is a mapping from $Expr$ to R . The expressions in the domain of RF represent states of the composition of C and φ , using additional variables to encode the “program counter” of C and the states of φ . Thus, given any ranking function $RF : Expr \rightarrow R$, we know C and φ implicitly.*

Definition 2 (Verification Condition) *Given a ranking function RF , we can effectively compute a formula called the verification condition of RF , denoted by $VC(RF)$, using an algorithm called *VC-Gen*.*

Ranking functions, verification conditions, and software certification are related intimately, as expressed by Fact 1. Note that we write $C \models \varphi$ to mean component C respects policy φ , and that a formula is *valid* if it is true under all possible variable assignments.

Fact 1 (Soundness) *For any component C and policy φ , if there exists a ranking function $RF : Expr \rightarrow R$ such that $VC(RF)$ is valid, then $C \models \varphi$.*

We will not go into a detailed proof of Fact 1, since it requires careful formalization of the semantics of C and φ . In addition, proofs of theorems that capture the same idea are presented elsewhere [Necula 1996, Chaki 2006].

Definition 3 (Certificate) *For any component C and policy φ , a certificate for $C \models \varphi$ is a pair (RF, Π) where $RF : Expr \rightarrow R$ is a ranking function over some finite set of ranks R , and Π is a proof of the validity of $VC(RF)$ in a sound proof system.*

Indeed, if such a certificate (RF, Π) exists, by the soundness of the proof system used to construct Π , we know that $VC(RF)$ is valid, and hence, by Fact 1, $C \models \varphi$. This style of certification, used in both PCC and CMC, has several tangible benefits:

- Any purported certificate (RF, Π) is validated by the following effective (i.e., automatable) procedure: (1) compute $VC(RF)$ using *VC-Gen* and (2) verify that Π is a correct proof of $VC(RF)$ using a proof checker.

- Necula and Lee demonstrated that this effective procedure satisfies a fundamental soundness theorem: any program with a valid certificate satisfies the policy for which the certificate is constructed [Necula 1996]. This fact is not altered even if the binary program, the proof certificate, or both are tampered with in any way. A binary program may exhibit different behavior in its modified form than in its original form. However, this new behavior will still be guaranteed to satisfy the published policy if its proof certificate is validated.
- The policy, VC-Gen, and proof-checking algorithms are public knowledge. Their mechanism does not depend in any way on secret information. The certificate can be validated independently and objectively. The soundness of the entire certification process is predicated solely upon the soundness of the underlying logical machinery, which is time tested, and the correctness of the trusted computing base (TCB), as discussed later.
- The computational complexity of the certification process is shouldered by the entity generating the certificate. In the case of software components, this entity is usually the component supplier who has the “burden of proof.”

Overall, the existence of a valid certificate implies that $C \models \varphi$ irrespective of the process by which the certified component was created or transmitted. This feature makes our certification approach extremely attractive for incorporating components derived from unknown and untrusted sources into safety-critical and mission-critical systems.

Resolution. The fundamental proof system used in our approach is called “resolution.” A resolution proof system is based on a single inference rule, also known as resolution. In the context of propositional Boolean formulas, the resolution rule can be stated as follows:

$$\frac{(p \vee q) \quad (p' \vee \neg q)}{(p \vee p')}$$

Intuitively, this rule means that if we have already inferred two clauses $(p \vee q)$ and $(p' \vee \neg q)$ containing the same variable q in positive and negative form, we can infer a new clause $(p \vee p')$ by combining the two original clauses and eliminating the “resolved” variable q .

In our approach, we use resolution as a strategy for proving the unsatisfiability of SAT formulas in conjunctive normal form (CNF). In this context, resolution is known to be both sound and complete. In other words, for any unsatisfiable SAT formula κ , there exists a finite sequence of applications of the resolution rule that start with κ and end with the inference of the empty clause (representing falsehood). Such a proof is also called a resolution proof, and state-of-the-art SAT solvers such as ZChaff [Zhang 2003] can emit resolution proofs upon proving a formula to be satisfiable. In addition, since resolution proofs involve just one simple inference rule, checkers for such proofs are easy to verify and trust.

3 Framework for Generating Certified Binaries

Figure 2 depicts the infrastructure for certified component binary generation that we have developed. Key elements are numbered for ease of reference and are correlated with the steps of the procedure described in this section. The flow of artifacts involved in generating a certified binary is indicated via arrows.

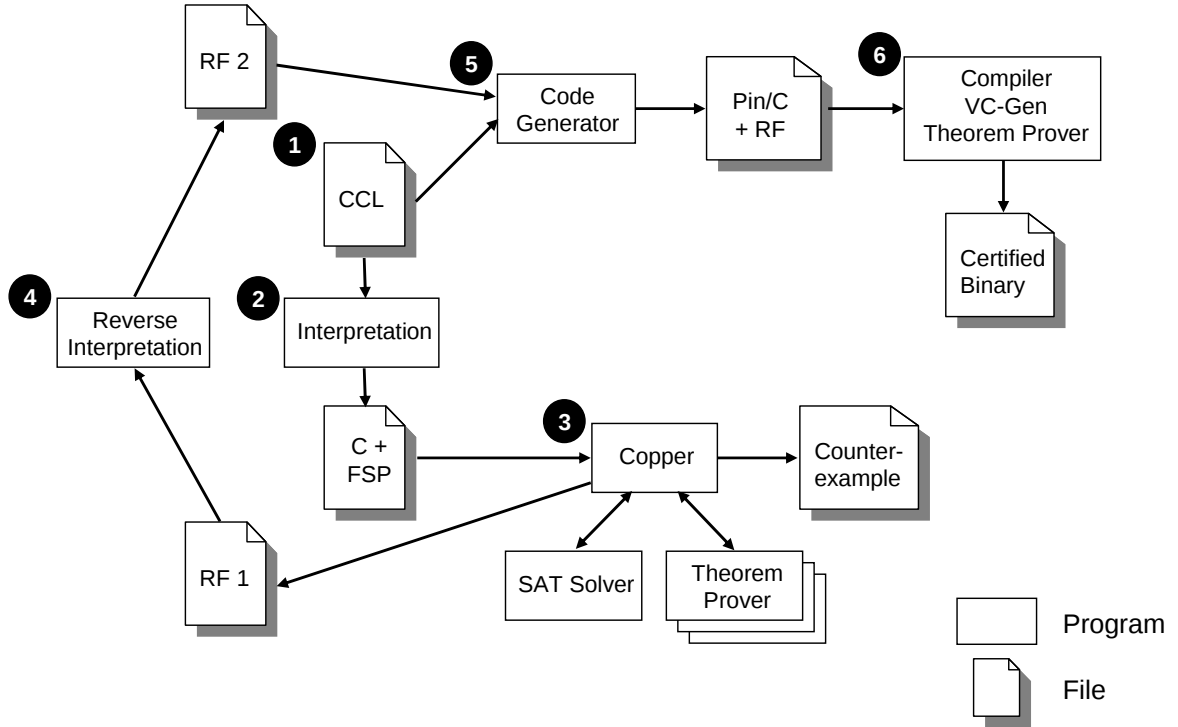


Figure 2: Framework for Generating Certified Binaries

3.1 THE PROCEDURE

The steps involved in generating certified component binaries are summarized below:

Step 1. A component is specified in CCL [Wallnau 2003]. CCL uses a subset of UML 2.0 statecharts that excludes features that are not particularly useful given the Pin component model as a target. The specification *Spec* contains a description of the component as well as the desired SE-LTL policy φ against which the component is to be certified.

Step 2. *Spec* is transformed (“interpreted” [Ivers 2002]) into a component *C* that can be processed by a model checker. *C* comprises a C program along with finite state machine specifications for the procedures it invokes. This step is implemented by augmenting the interpretation for the SEI’s ComFoRT [Ivers 2004] reasoning framework so that *C* contains additional information relating its line numbers, variables, and other data structures to those of *Spec*. This information is crucial for the subsequent reverse interpretation of ranking functions in Step 4.

Step 3. C is input into Copper, a state-of-the-art certifying software model checker. Copper [Chaki 2005a] was originally developed as part of ComFoRT and interfaces with theorem provers (TP) and Boolean satisfiability solvers (SAT). The output of Copper is either a counterexample (CE) to the desired policy φ or a ranking function $RF1 : Expr \rightarrow R$ over some set of ranks R such that $VC(RF1)$ is valid.

Step 4. The certificate $RF1$ only certifies C (the result of the interpretation) against the policy φ . It is reverse interpreted into a certificate $RF2 : Expr \rightarrow R$ such that $VC(RF2)$ is valid. The reverse interpretation is enabled by the additional information generated during interpretation to connect $Spec$ with C (see Step 2).

Step 5. $Spec$ and $RF2$ are transformed into Pin/C component code that can be compiled and deployed in the Pin runtime environment (RTE) [Hissam 2005]. Pin/C code generation without ranking functions had been developed as part of PACC. We augment this code-generation step to also create a ranking function, using $RF2$, and embed it in the generated code. In essence, we transform the ranking function, along with the component, from one format (CCL) to another (Pin/C).

Step 6. The final step consists of three distinct substeps.

- [a] The component with the embedded ranking function is compiled from Pin/C to binary form. In our implementation, we use GCC⁴ (that targets the PowerPC instruction set) to achieve this goal. We refer to the ranking function embedded in the binary as $RF3$.
- [b] We compute $VC(RF3)$ using VC-Gen.
- [c] We obtain a proof Π of $VC(RF3)$ using a proof-generating theorem prover. For our implementation, we use a SAT-based theorem prover. In essence, we convert $\neg VC(RF3)$ (i.e., the logical negation of $VC(RF3)$) to a Boolean formula ϕ . We then check if ϕ is unsatisfiable using ZChaff [Zhang 2003]. If $VC(RF3)$ is valid, the resolution proof produced by ZChaff serves as Π . The use of SAT enables us to obtain extremely compact proofs in practice [Chaki 2006]. Finally, the certificate (RF, Π) along with the binary is produced as the end result—the certified binary for $Spec$.

Steps 1-6 are elaborated in Sections 4-6.

3.2 TRUSTED COMPUTING BASE

Certain artifacts must be trustworthy in order for our approach to be effective. In essence, the trusted computing base (TCB) is composed of

1. VC-Gen
2. the procedure for converting $\neg VC(RF3)$ to ϕ
3. the procedure for checking that Π refutes ϕ

All of these procedures are computationally inexpensive and can be implemented by relatively small programs. Thus, they are more trustworthy (and more verifiable) than the rest of the programs shown in Figure 2. For instance, they can be implemented in proof-based systems like ACL2.⁵

⁴ For more information, go to <http://gcc.gnu.org/>.

⁵ For more information, go to <http://www.cs.utexas.edu/users/moore/acl2/>.

Specifically, the programs no longer required to be in the TCB are the interpreter, the certifying model checker, the reverse-interpreter, the code generator, the compiler, and the theorem prover. These tools are all quite complex, and eliminating them from the TCB raises considerably the degree of confidence of our certification method.

The fact that the model checker itself need not be trusted is not (initially) easily grasped, but this astonishing fact is so fundamentally important that further elaboration is justified. In fact, this concept can be demonstrated in the infrastructure described in this report:

- The Copper model checker uses the Simplify [Nelson 1980] theorem prover to create program abstractions. Because Simplify regards integer types as unbounded numbers, an invariant of the form $x < x + 1$ might be generated. Of course, in unbounded arithmetic, this invariant is trivially valid. However, this invariant is *not* valid when we consider bit-level semantics of integer types represented on a computer (i.e., integer overflow). In short, Copper can produce unsound results if bit-level semantics of programs will influence the satisfaction of a desired policy.
- The process for constructing $\neg VC(RF3)$ and for converting this to ϕ is part of the TCB. The conversion to ϕ also encodes bit-level semantics in SAT form. These semantics are missing from Simplify, which explains why the model checker may have produced an unsound result.
- The proof of ϕ is generated by a SAT solver. At this point it is likely that the SAT solver will fail to produce a proof of ϕ . However, the SAT solver (which we do not trust) may generate a faulty resolution proof of ϕ .
- Even if the model checker and the SAT solver are faulty, the proof produced by the SAT solver is checked independently against the published rules for resolution proofs. The proof checker must be part of the TCB, but it is much smaller and simpler when compared to the model checker and SAT solvers.

The above scenario demonstrates that the model checker and SAT solver used to generate abstractions or proofs of policy satisfaction need not be trusted. Only the relatively simpler processes of generating ϕ and checking resolution proofs that ϕ is unsatisfiable need to be trusted.

How the TCB is demarcated and how its size and complexity are reduced are important theoretical and practical concerns for future applications of PCC. There are several approaches to these concerns. For example, “foundational” PCC [Appel 2001] aims to reduce the TCB to its bare minimum of logic foundations. We adopt the more systems-oriented approach pioneered by Necula and Lee, which does not seek a pure foundation but rather seeks to achieve a practical compromise [Schneck 2002]. Even this more “pragmatic” approach can achieve good results. In our own implementation, the TCB is more than 15 times smaller in size (30 KB vs. 450 KB) than the rest of the infrastructure (which includes code generators, model checkers, and ancillary theorem provers such as Simplify).

4 Certifying Model Checking

The infrastructure for certifying model checking corresponds to Steps 1-4 from Figure 2. We begin with an overview of iterative predicate abstraction and refinement, the core paradigm used by Copper to verify C programs.

4.1 ITERATIVE PREDICATE ABSTRACTION AND REFINEMENT

In iterative abstraction refinement, conservative models of the C program are constructed via predicate abstraction, verified, and then refined iteratively until either the verification succeeds or a real counterexample is found. We illustrate this paradigm with a simple example.

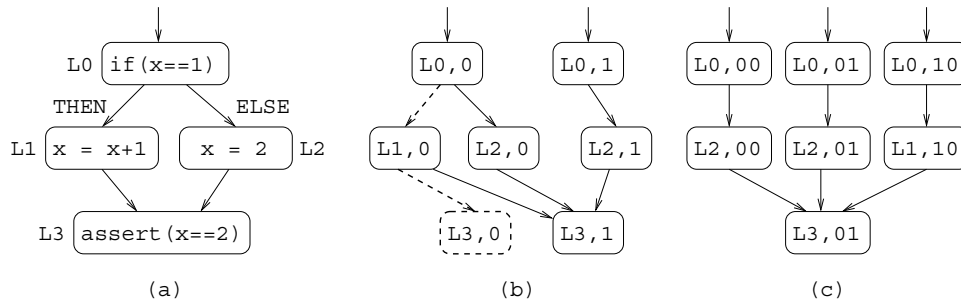


Figure 3: Example of Software Verification Via Iterative Refinement

Consider the C program represented by the CFG shown in Figure 3(a). The program begins with a branch statement at location L0. If the branch condition holds, the assignment at location L1 is executed. Otherwise, the assignment at location L2 is executed. Finally, a call to the procedure `assert` is made at location L3. The safety policy to be verified is that the argument to `assert` is always non-zero.

Predicates and Concretization. As mentioned before, Copper creates conservative models of the C program using a technique called predicate abstraction. The result of predicate abstraction is parameterized by a finite set of predicates, where each predicate is a side-effect-free C expression. Let $\mathcal{P}$ be a set of predicates. Then a valuation of $\mathcal{P}$ is a mapping from $\mathcal{P}$ to $\{\text{TRUE}, \text{FALSE}\}$. We use a sequence of zeroes and ones to denote valuations, as follows. Let $\mathcal{P} = \{x == 1, x == 2\}$ be a set of predicates. Then the valuation $V = \{0, 1\}$ maps predicate $(x == 1)$ to FALSE and $(x == 2)$ to TRUE. The *concretization* of a valuation V for a set of predicates $\mathcal{P}$ is denoted by $\gamma(V)$ and is defined as follows:

$$\gamma(V) = \bigwedge_{p \in \mathcal{P}} p^{V(p)}$$

where $p^{\text{TRUE}} = p$ and $p^{\text{FALSE}} = \neg p$. For example, if $\mathcal{P} = \{x == 1, x == 2\}$ and $V = \{0, 1\}$, then $\gamma(V) = (x != 1) \ \&\& \ (x == 2)$. Note that we use C syntax for logical operators to express concretizations as side-effect-free C expressions. Also, for any two distinct valuations V and V' , $\gamma(V)$ and $\gamma(V')$ are logically disjoint.

Predicate Abstraction. The result of predicate abstraction with a set of predicates $\mathcal{P}$ is a finite state machine model M such that... (1) the states of M are of the form (l, V) , where l is a statement location in the C program and V is a valuation of $\mathcal{P}$ and (2) the transitions between states are defined existentially. The key idea behind predicate abstraction is that a predicate valuation is an abstract

description of a (possibly infinite) collection of concrete memory states. Specifically, the valuation $V = \{0, 1\}$ for $\mathcal{P} = \{x == 1, x == 2\}$ represents the set of all memory states of the program where the value of variable x is 2.

Therefore, an abstract state (l, V) of the model M represents the set of all concrete states of the C program such that: (1) the statement at location l is to be executed next and (2) the current memory state is described by V . Let us write $CS(l, V)$ to denote the set of all concrete states represented by the abstract state (l, V) . Then, M has a transition from (l, V) to (l', V') if there exist concrete states $s \in CS(l, V)$ and $s' \in CS(l', V')$ such that program's execution from s leads to s' . While constructing M , Copper uses a theorem prover, such as Simplify, to decide if a transition should exist between any two abstract states.

Copper creates the first model with a set of predicates derived from the policy. In the example from Figure 3(a), this set of predicates is $\mathcal{P}_1 = \{x == 2\}$. The abstract model M_1 created with $\mathcal{P}_1$ is shown in Figure 3(b). Each state of M_1 is labeled with a pair consisting of a location and a predicate valuation. Upon model checking M_1 , Copper finds a counterexample CE , denoted by the dashed arrows in Figure 3(b), that leads from an initial state to a state where the predicate $(x == 2)$ is FALSE.

Copper then determines whether CE corresponds to a real execution of the C program. However, in this case, CE turns out to be *spurious* (i.e., an artificial behavior introduced by the conservative nature of the abstraction). Copper then adds new predicates to eliminate CE and reconstructs the model via predicate abstraction. In our example, Copper adds the predicate $(x == 1)$ so that the new predicate set is $\mathcal{P}_2 = \{x == 1, x == 2\}$. The result M_2 of predicate abstraction with $\mathcal{P}_2$ is shown in Figure 3(c). Model M_2 has no reachable states where the predicate $(x == 2)$ is FALSE and hence is verified successfully. Copper then generates a witness to successful verification, as described in Section 4.4.

4.2 CONSTRUCTION AND COMPOSITION LANGUAGE

We now give a brief overview of the construction and composition language (CCL), a simple composition language used to describe how components behave and how components are wired together into assemblies for deployment [Wallnau 2003]. For this work, we certify the behavior of individual components, and we focus on the use of CCL to specify the behavior of software components. Figure 4 shows a CCL specification for a simple component. (Note that the specification corresponds to Step 1 from Figure 2.) This component reacts to stimuli from its environment on its `incr` sink pin by incrementing an internal counter (up to a maximum and then resetting to a minimum) and informing its environment of the new value on its `value` source pin.

In CCL, behavior is described in terms of potentially concurrent units of computation called *reactions*, which describe how a component responds to incoming stimulation on its sink pins and under what circumstances it will initiate interactions on its source pins. The semantics of the state machine provided for each reaction is based on the UML 2.0 semantics of statecharts. Aside from the obvious syntactic differences (CCL text as shown in Figure 4 vs. UML graphical notation as shown in Figure 5), CCL differs from statecharts in two important ways:

1. CCL does not permit some concepts defined in UML statecharts—most significantly hierarchical states and concurrent substates within a reaction.
2. CCL provides more specific semantics for elements of the UML standard that are identified as semantic variation points (e.g., the queuing policy for events queued for consumption by a state machine). These refined semantics are based on the execution semantics of the Pin component technology, the target of our code generator.

```

const int min = 0;
const int max = 6;

component comp () {
  sink asynch incr ();
  source asynch value (produce int v);

  threaded react R (incr, value) {
    int i = min;

    start -> idle { }

    idle -> incrementing {trigger ^incr;}
    incrementing -> idle {trigger $value; action $incr();}

    state incrementing {
      if (i < max) {
        i++;
      } else {
        i = min;
      }
      ^value(i);
    }
  } // end of react R
} // end of component comp

```

Figure 4: CCL Specification for a Simple Component

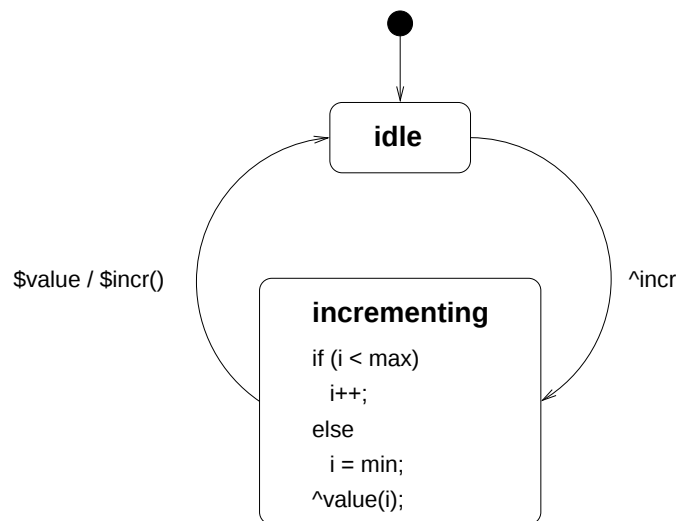


Figure 5: Graphical Depiction of the Component's State Machine

4.3 INTERPRETATION FROM CCL TO C

To model check CCL specifications, we generate an equivalent interpretation in a form suitable for use with Copper, a software model checker. The interpreted form is a C implementation of the component's state machine, an excerpt of which is shown below (note that this C implementation corresponds to the output of Step 2 from Figure 2).

```
void R() {
    int curState = _R__START_;
    int nextEvent;
    int R__i = min;

    CCL_NODE(16777255);
    __COPPER_NONDET__();
    curState = _R__idle_;

    label_R__idle_:
        // no state actions
        label_listening_R__idle_:
            nextEvent = fsp_R_externalChoice();
            CCL_NODE(16777268);
            if (nextEvent == __R__incr__) {
                // Consume interaction on R__incr
                // no transition action
                CCL_NODE(16777272);
                curState = _R__incrementing_;
                goto label_R__incrementing_;
            }
            goto label_listening_R__idle_;

    label_R__incrementing_:
        CCL_NODE(106);
        if (R__i < max) {
            CCL_NODE(94);
            R__i++;
        } else {
            CCL_NODE(104);
            R__i = min;
        }
        CCL_NODE(116);
        __COPPER_HANDSHAKE__("begin__R__value");
        R__value__v = R__i;
        __COPPER_HANDSHAKE__("begin__R__value__done");
        CCL_NODE(16777273);
        __COPPER_NONDET__();
        CCL_NODE(67);
        __COPPER_HANDSHAKE__("end__R__value");
        CCL_NODE(79);
        __COPPER_HANDSHAKE__("end__R__incr");
        CCL_NODE(16777283);
        curState = _R__idle_;
        goto label_R__idle_;

} // end of R
```

In the interpreted form, each state of the specification state machine is implemented in a correspondingly labeled program block: guards are represented by `if` statements, transitions are

completed using `goto` statements, and so on. The equivalence is straightforward, particularly given CCL's use of C syntax for actions. Two elements that are less intuitive are the representation of events used for interaction (communication) between components and annotations used to facilitate reverse interpretation (expressing model checking results in terms of the original CCL specification instead of the interpreted C program).

Communication between concurrent units (representations of interacting components) in Copper is primarily handled using event semantics based on finite state processes (FSP) [Magee 2006]. Our interpretation uses events to model message-based interactions between components in the Pin component technology. In Pin, interactions occur in synchronous or asynchronous modes, and the initiation and completion of an interaction are differentiated syntactically by a `^S` for initiation on a pin `S` or a `$$` for completion on a pin `S`. These phenomena are mapped to FSP-style events as part of the interpretation. For example, initiation of an interaction over a source pin (`value`) is represented by a `begin_value` event. This event is denoted in the interpreted C program using the `__COPPER_HANDSHAKE__()` function.

Representing a choice among several events, however, is more difficult. For example, a component's willingness to engage in an interaction over any of several sink pins (i.e., pull the next message from its queue and respond accordingly) corresponds to a willingness to synchronize over one of several FSP-style events. This concept is not as easily represented in C, and here we use Copper's ability to provide specifications of a function's behavior. We insert a call to an `fsp_externalChoice()` function and provide a specification of that function's behavior as an FSP process that allows a choice among a specific set of events and returns an integer indicating the event with which the process synchronized.

The annotations used to simplify reverse interpretation are inserted via `CCL_NODE(x)` function calls. The parameter passed to each such call denotes a node in the CCL abstract syntax tree (AST) for the CCL specification that corresponds to a C statement that follows the annotation. These functions are known to Copper and normally stripped from the program prior to verification. When used for certifying model checking, however, Copper retains the parameter values and includes them in the ranking functions produced upon successful verification.

4.4 RANKING FUNCTIONS GENERATED BY COPPER

After a C program (with FSP specifications) is generated via interpretation, Copper is used to verify whether each policy (specified as an SE-LTL formula) is satisfied (Step 3 from Figure 2). As mentioned before, Copper uses automated iterative abstraction refinement to verify its target C program against the desired policy. Normally, model checkers generate counterexamples when they detect the violation of a policy. The counterexample is a machine-checkable *witness* to this failure. In contrast, certifying model checkers also generate a witness to success. Specifically, in our certifying model checker Copper, this witness takes the form of a ranking function. We now describe the structure of the ranking functions produced by Copper in more detail.

Let M be the model verified successfully. Recall that each state of M is of the form (l, V) where l is a location in the C program, and V is a valuation of the set of predicates used to construct M . Copper emits the ranking function as a set of triples of the form $((l, I), s, r)$ where: (1) I is an invariant (i.e., the concretization of a predicate valuation V such that (l, V) is a reachable state of M),⁶ (2) s is a state of the Büchi automaton corresponding to the policy, and (3) r is a rank. The procedure for constructing an appropriate ranking function has been presented elsewhere, so we do not describe it further here [Chaki 2006].

Recall, from Definition 1, that a ranking function is a mapping from expressions to ranks. Each triple $((l, I), s, r)$ produced by Copper corresponds to an entry in this mapping as follows. Let PC and SS

⁶ Strictly speaking, an invariant at location l is the disjunction of the concretizations of all predicate valuations V such that (l, V) is a reachable state of M . We use a slightly looser definition of invariant for simplicity.

be special variables representing the program location (i.e., program counter) and the specification state, respectively. Then, the triple $((l, I), s, r)$ denotes a mapping in the ranking function from the expression $I \ \&\& \ (PC == l) \ \&\& \ (SS = s)$ to the rank r . Note that, for any two triples $((l, I), s, r)$ and $((l', I'), s', r')$ produced by Copper, either $l \neq l'$ or I and I' are disjoint (since they are the concretizations of two distinct predicate valuations). Hence, the ranking function produced is always well formed.

Figure 6 shows an excerpt from the ranking function generated for our example CCL specification and a policy asserting that $\min \leq i \leq \max$ is always true (the input to Step 4 from Figure 2). Each line denotes a triple $((l, I), s, r)$. The first field is the CCL AST node number, corresponding to the location l . The second and third fields (which, in the excerpt, are always 8 and 0) correspond to the policy automaton state s and the rank r , respectively. The last field is the invariant I .

```
104 : 8 : 0 [(-1 < P0::R__i ), (P0::R__i < 7 )]
106 : 8 : 0 [(P0::R__i < 7 ), (-2 < P0::R__i ), (P0::R__i != -1 )]
116 : 8 : 0 [(-1 < P0::R__i ), (P0::R__i < 7 )]
```

Figure 6: Ranking Function in Terms of the C Program

The final step in our infrastructure for certifying model checking is to relate the ranking function back to the original CCL specification. This step is achieved via a process of mapping elements from the interpreted C program back to CCL elements. For example, variable names are “demangled” and replaced with references to AST node numbers (of the form $@n$), and predicates relating to variables that were introduced during interpretation are stripped or remapped to the appropriate CCL concepts. An example of this corresponding to the “raw” ranking function from Figure 6 is shown in Figure 7 (the output of Step 4 from Figure 2).

```
37 : 104 : 8 : 0 : [(-1 < @46 ) && (@46 < 7 )]
37 : 106 : 8 : 0 : [(-2 < @46 ) && (@46 != -1 ) && (@46 < 7 )]
37 : 116 : 8 : 0 : [(-1 < @46 ) && (@46 < 7 )]
```

Figure 7: Ranking Function in Terms of the CCL Specification

At the conclusion of certifying model checking, if a component is known to satisfy all of its policies, the tools produce evidence to that effect in the form of a ranking function expressed in terms of AST nodes for the component’s CCL specification.

5 Certified Source-Code Generation

The infrastructure for generating certified source code corresponds to Step 5 from Figure 2. We begin with a component specification expressed in CCL and a ranking function expressed in terms of AST nodes corresponding to that CCL specification.

From previous work, we have a code generator for CCL that generates C code targeted for deployment in the Pin component technology (Pin/C). As such, in addition to the kind of instructions used to represent the state machine for verification, platform-specific instructions are included for integrating with the Pin component model, (e.g., when and how to examine incoming message traffic), for tasks such as marshaling data to be included in an outgoing message, and for error checking.

To support certified code generation, we extended this code generator to embed invariants from the ranking function in the generated Pin/C code. The key decision was choosing how to embed this information so that a correlation would be maintained between the location of these invariants in the Pin/C code and the location of equivalent information in assembly code resulting from compiling the Pin/C code.

We chose a convention in which we insert a pair of function calls in the Pin/C code prior to the location associated with each invariant. The invariant itself is used as the argument to the second function of the pair. When such code is compiled, pairs of recognizable assembly call instructions appear in the assembly code, and the instructions necessary to represent the invariant appear between these calls. This convention and the mapping between C and assembly code are shown in Figure 8. Specifically, in the assembly portion of the figure, Line 1 denotes the call to `__begin__()`, Line 14 denotes the call to `__inv__()`, and lines 2–13 denote the computation of the value of `(n >= 0) && (n < 10)` with the result being stored in register `r3`. Observe that variable `n` is stored in memory location `r31 + 8` (as seen in lines 4 and 7).

<code>__begin__();</code>	1:	<code>bl __begin__</code>
	2:	<code>li %r0,0</code>
	3:	<code>stw %r0,16(%r31)</code>
	4:	<code>lwz %r0,8(%r31)</code>
	5:	<code>cmpwi %cr7,%r0,0</code>
	6:	<code>blt %cr7,.L5</code>
	7:	<code>lwz %r0,8(%r31)</code>
	8:	<code>cmpwi %cr7,%r0,9</code>
	9:	<code>bgt %cr7,.L5</code>
	10:	<code>li %r0,1</code>
	11:	<code>stw %r0,16(%r31)</code>
	.L5:	
	12:	<code>lwz %r3,16(%r31)</code>
	13:	<code>crxor 6,6,6</code>
<code>__inv__((n >= 0) && (n < 10));</code>	14:	<code>bl __inv__</code>

Figure 8: Invariants in Pin/C Code (Left) and Assembly Code (Right)

We extended the Pin/C code generator to insert these pairs of calls at any location for which the ranking function provides invariants. The code generator adds an additional predicate to each invariant found in the ranking function, an encoding of the current state of the state machine. This piece of information is essential to mapping the control flow in the Pin/C program to the corresponding state machine, particularly given that the calling convention for the C function implementing the state machine differs between the interpreted C function and the Pin/C function, as described next.

In Pin, the function implementing the state machine is called by its container (which, among other things, enforces the life cycle defined by Pin) whenever a new message is available for consumption. The function then runs until it is ready to wait for the next message to arrive, at which point it returns control to the container. This pattern is equivalent in effect to the interpreted C program's use of the `fsp_externalChoice()` convention for interaction with the component's environment.

The key difference is that in the interpreted C program, the entire state machine is executed within a single function. In the Pin/C program, the function is executed repeatedly, and global information is used to keep track of the current state for the next invocation of the function. This difference also requires that the code generator insert invariants at specific points in the program that are derived from those found in the ranking function.

For example, at the conclusion of a block of code that could transition the state machine to one of several successor states, we insert as an invariant the disjunction of the invariants known to hold prior to the beginning of each potential successor state. An example of this is shown at the end of the following excerpt from the generated Pin/C code for our simple example.

```
else if ( _THIS->R_CURRENT_STATE == 1 ) {
    __begin__();
    __inv__((__pcc_claim__ == 0 && __pcc_specstate__ == 8 && __pcc_rank__ == 0 &&
        ((-2 < _THIS->R_i ) && ( _THIS->R_i != -1 ) && ( _THIS->R_i < 7 )) &&
        _THIS->R_CURRENT_STATE == 1))); /* 52 */

    if (pMessage->sinkPin == 0 /* ^incr */ ) {
        __sink_0__(); /* ^incr */

        // no action on idle->incrementing transition

        _THIS->R_CURRENT_STATE = 2;
        __begin__();
        __inv__((__pcc_claim__ == 0 && __pcc_specstate__ == 8 && __pcc_rank__ == 0 &&
            ((-2 < _THIS->R_i ) && ( _THIS->R_i != -1 ) && ( _THIS->R_i < 7 )) &&
            _THIS->R_CURRENT_STATE == 2))); /* 57 */

        // incrementing action:
        __begin__();
        __inv__((__pcc_claim__ == 0 && __pcc_specstate__ == 8 && __pcc_rank__ == 0 &&
            ((-2 < _THIS->R_i ) && ( _THIS->R_i != -1 ) && ( _THIS->R_i < 7 )))) &&
            _THIS->R_CURRENT_STATE == 2); /* 106 */
        if ( _THIS->R_i < G_max ) {
            _THIS->R_i++;
        }
        else {
            __begin__();
            __inv__((__pcc_claim__ == 0 && __pcc_specstate__ == 8 &&
                __pcc_rank__ == 0 && ((-1 < _THIS->R_i ) && ( _THIS->R_i < 7 )))) &&
                _THIS->R_CURRENT_STATE == 2); /* 104 */
            _THIS->R_i = G_min;
        }

        __begin__();
        __inv__((__pcc_claim__ == 0 && __pcc_specstate__ == 8 && __pcc_rank__ == 0 &&
            ((-1 < _THIS->R_i ) && ( _THIS->R_i < 7 )))) &&
            _THIS->R_CURRENT_STATE == 2); /* 116 */
        __source_0__(); /* value */
    }
}
```

```

//-----Marshall each actual param on interaction-----
__marshDx = 0;
__marshInt = _THIS->R_i;
memcpy(&MessageOut.data[__marshDx], &__marshInt, sizeof(PIN_INT));
__marshDx += sizeof (PIN_INT);
//-----Call asynchronous IPC mechanism-----
if (!sendOutSourcePin( /* ^value() */ pReaction, 0, &MessageOut,
    (short) (sizeof(MessageOut.data)),
    IPCPORT_WAITFOREVER /* TBD property */) ) {
    notifyController(pReaction->pInstance, CONTROLLER_UNKNOWN_ERROR,
        "error in SendOutSourcePin");
    ...
}
//-----
}
__begin__();
__inv__((__pcc_claim__ == 0 && __pcc_specstate__ == 8 && __pcc_rank__ == 0 &&
    ((-2 < _THIS->R_i ) && (_THIS->R_i != -1 ) && (_THIS->R_i < 7 )) &&
    _THIS->R_CURRENT_STATE == 2))/* 57 */
|| ((__pcc_claim__ == 0 && __pcc_specstate__ == 8 && __pcc_rank__ == 0 &&
    ((-2 < _THIS->R_i ) && (_THIS->R_i != -1 ) && (_THIS->R_i < 7 )) &&
    _THIS->R_CURRENT_STATE == 1))/* 52 */
);
}

```

At the conclusion of certified source-code generation, we have C source code that includes the invariants necessary for generating a proof that the binary form of this component satisfies the desired policy. An important point to note is that the generated certified source code contains at least one call to `__begin__()` and `__inv__(...)` inside every loop. This convention is crucial for effective computation of the certified binary, as presented in the next section, without having to supply loop invariants.

6 Certified Binary Generation

In this section, we describe the process of obtaining the certified binary code. To this end, we present the procedure for constructing the two components of the certified binary: the binary itself and a certificate that is the proof of a verification condition.

Binary Construction. Recall from the previous section that we have C source code that includes the invariants necessary for generating a proof that the binary form of this component satisfies the desired policy. The certified binary is then obtained by compiling this C source code using any standard compiler. In our implementation, we used GCC targeted at the PowerPC instruction set for the compilation step of our procedure.

The binary generated by the compiler contains assembly instructions, peppered with `__begin__()` and `__inv__(...)` calls. Let us refer to an assembly fragment starting with a call to `__begin__()` and extending up to the first following call to `__inv__(...)`, as a *binary invariant*. Note that in any binary invariant, the code between the calls to `__begin__()` and `__inv__(...)` effectively computes and stores the value of the argument being passed to `__inv__(...)` in register `r3`. For example, Figure 9 shows a snippet with the source code on the left, and the corresponding PowerPC assembly on the right. The assembly fragment has two binary invariants, one spanning lines 1–9 and the other spanning lines 13–21.

<code>__begin__();</code>	1: <code>bl __begin__</code>
	2: <code>lis %r9,UCOS_LOCK@ha</code>
	3: <code>lwz %r0,UCOS_LOCK@l(%r9)</code>
	4: <code>cmpwi %cr7,%r0,0</code>
	5: <code>mfcrr %r0</code>
	6: <code>rlwinm %r0,%r0,31,1</code>
	7: <code>mr %r3,%r0</code>
	8: <code>crxor 6,6,6</code>
<code>__inv__(UCOS_LOCK == 0);</code>	9: <code>bl __inv__</code>
<code>UCOS_LOCK = 1;</code>	10: <code>lis %r9,UCOS_LOCK@ha</code>
	11: <code>li %r0,1</code>
	12: <code>stw %r0,UCOS_LOCK@l(%r9)</code>
<code>__begin__();</code>	13: <code>bl __begin__</code>
	14: <code>lis %r9,UCOS_LOCK@ha</code>
	15: <code>lwz %r0,UCOS_LOCK@l(%r9)</code>
	16: <code>cmpwi %cr7,%r0,1</code>
	17: <code>mfcrr %r0</code>
	18: <code>rlwinm %r0,%r0,31,1</code>
	19: <code>mr %r3,%r0</code>
	20: <code>crxor 6,6,6</code>
<code>__inv__(UCOS_LOCK == 1);</code>	21: <code>bl __inv__</code>

Figure 9: Invariants and Code in Pin/C (Left) Assembly (Right)

Certificate Construction. To construct the certificate, we first construct the verification condition VC , which is done one binary invariant at a time. Specifically, for each binary invariant β , we compute the verification condition for β , denoted by $VC(\beta)$. Let BI be the set of all binary invariants in our binary. Then, the overall verification condition VC is defined as follows:

$$VC = \bigwedge_{\beta \in BI} VC(\beta)$$

The technique for computing $VC(\beta)$ is based on computing weakest preconditions, the semantics of the assembly instructions, and the policy that the binary is being certified against. It is similar to the VC-Gen procedure used in PCC. The main difference is that our procedure is parameterized by the policy and is thus general enough to be applied to any policy expressible in SE-LTL. In contrast, the VC-Gen procedure used in PCC has a “hard-wired” safety policy, namely, memory-safety. It is also noteworthy that our procedure does not require loop invariants, since every loop in the binary contains at least one binary invariant.

Once we have VC , the certificate is obtained by proving VC with a proof-generating theorem prover. In this context, we leverage our previous work on using a theorem prover based on Boolean satisfiability (SAT) [Chaki 2006]. This approach yields extremely compact certificates when compared to existing non-SAT-based proof-generating theorem provers. In addition, it enables us to be sound with respect to bit-level C semantics, which is crucial when certifying safety-critical software.

The above procedure is complex in its details but simple in overall concept. To reiterate at the concept level: Given a binary B and an associate certificate C , we first compute the verification condition VC using the technique described above. We then check that C is a correct proof of the validity of VC . Validation succeeds if and only if C turns out to be a proper proof of VC .

A final point to note is that once a certified binary has been validated successfully, the embedded binary invariants are stripped off before the binary is actually deployed, which is crucial for both correctness (since what we really certify is the binary *without* the invariants) and performance.

7 Related Work

PCC was proposed by Necula and Lee for certifying memory safety policies on binaries [Necula 1997, Necula 1996, Necula 1998b]. PCC works by hard-coding the desired safety policies within the machine instruction semantics. In contrast, our approach works at the specification level and encodes the policy as a separate automaton. Foundational PCC attempts to reduce the trusted computing base of PCC to include only the foundations of mathematical logic [Appel 2001, Hamid 2002]. In contrast, the focus of our work is to extend the ideas behind PCC toward the generation of certified binaries from software component specifications. Bernard and Lee propose a new temporal logic to express PCC policies for machine code [Bernard 2002]. However, we use temporal logic to specify claims at the level of component specifications. While we leverage SAT technology to generate compact proofs, non-SAT-based techniques for minimizing PCC proof sizes have also been investigated [Necula 1998a, Necula 2001]. Whalen and colleagues describe a technique for synthesizing certified code [Whalen 2002]. They augment the AUTOBAYES synthesizer to add annotations based on “domain knowledge” to the generated code. Their approach is not based on CMC and generates certified source code rather than binaries.

Certifying model checkers emit an independently verifiable certificate of correctness when a temporal logic formula is satisfied by a finite state model [Namjoshi 2001, Kupferman 2004]. Namjoshi has proposed a two-step technique for obtaining proofs of Mu-Calculus policies on infinite state systems [Namjoshi 2003]. In the first step, a proof is obtained via certifying model checking. In the second step, the proof is “lifted” through an abstraction. Namjoshi’s approach is still restricted to certifying source code while our work aims for low-level binaries.

Iterative refinement has been applied successfully by several software model checkers such as SLAM [Ball 2001], BLAST [Henzinger 2002b], and MAGIC [Chaki 2004a]. While SLAM and MAGIC do not generate any proof certificates, BLAST implements a method for lifting proofs of correctness [Henzinger 2002a]. However, BLAST’s certification is limited to source code and purely safety properties. Assurance about the correctness of binaries can also be achieved by proving the correctness of compilers or via translation validation [Pnueli 1998]. Both of these techniques assume that the original source code or specification is correct. They are difficult to do and are not yet widely adopted. In contrast, our approach requires no such correctness assumptions.

In previous work, we developed an expressive linear temporal logic SE-LTL that can be used to express both safety and liveness claims of component-based software [Chaki 2004b]. In the work reported here, we used SE-LTL to express certifiable policies. Also previously, we developed an infrastructure to generate compact certificates for *C source code* against SE-LTL claims in an automated manner [Chaki 2005b]. There, the model checker is used to generate invariants and ranking functions that are required for certificate and proof construction. Compact proofs were obtained via state-of-the-art Boolean satisfiability (SAT) technology [Chaki 2006]. In the current work, we extend this framework to generate certified *binaries* from component specifications. Finally, we build on the PACC infrastructure for analyzing specifications of software component assemblies and generating deployable machine code for such assemblies.

8 Experimental Results

We implemented a prototype of our technology and experimented with two kinds of examples. First, we created a CCL specification of a component that manipulates an integer variable and a policy that the variable never becomes negative. Our tool was able to successfully prove, and certify at the assembly code level, that the implementation of the component does indeed satisfy the desired claim. The CCL file size was about 2.6 KB, while the generated Pin/C code was about 20 KB. In contrast, the assembly code was about 110 KB, while the proof certificate size was just 7.7 KB. The entire process took about five minutes with modest memory requirements.

To validate the translation of a certified C component to a certified binary (Step 7 in Figure 1), we conducted additional experiments with Micro-C, a lightweight operating system for embedded real-time applications. The OS source code consists of about 6,000 lines of C (97 KB) and uses a semaphore to ensure mutually exclusive access to shared kernel data structures. Using our approach, we were able to certify that all kernel routines follow the proper locking order when using the semaphore. In other words, the semaphore is always acquired and released alternately, and thus deadlock is avoided. The total certification time was about one minute, and the certificate size was about 11 KB, or roughly 11% of the operating system source code size.

We also experimented with the C implementation of the “tar” program in the Plan 95 operating system. Specifically, using our approach, we certified that a particular buffer will never overflow when the program is executed. The source code was manually annotated in order to generate the appropriate proof certificates. While our experiments showed that our approach is viable, we believe that a more robust implementation and more realistic case studies are necessary to transition our technique to a broader user base.

9 Conclusion

In this report, we presented an automated approach for generating certified binaries from software component specifications. Our approach is based on, and combines the strengths of, two existing paradigms for formal software certification—PCC and CMC. It also demonstrates that a model-driven approach can be combined effectively with formal certification methodologies in a way that allows us to trust the code produced without having to trust the tools that produced it. In addition, we developed and experimented with a prototypical implementation of our approach.

Our implementation and overall approach do have limitations, which we classify using the following four categories:

Deferred Features. Some of the missing features from our implementation are not difficult conceptually but are best deferred until a target environment for the approach has been selected. For example, we did not define the format of certified binaries—in particular how the proof object is packaged with executable code.

Technical Limitations. Some limitations of our implementation do not depend on target deployment but rather are limitations of our own tooling that could be addressed with time and effort. For example, CCL currently supports only a primitive assortment of types, and, hence, the implementation supports a limited range of C-language features (e.g., pointers, structs, and arithmetic types other than int and float are not supported). Also, we have not implemented our own proof checker or SAT formula generator, even though these are key elements of a TCB. Instead, we rely on (in principle) untrusted publicly available implementations. However, both of these tools are relatively simple to implement. Also, Copper is only able to generate ranking functions that involve a finite and strictly ordered set of ranks and thus can certify a restricted set of programs. More general ranking functions are generated by other tools such as Terminator.⁷

Intrinsic Limitations. The model-driven approach that we demonstrate differs in several fundamental ways from more established PCC approaches. Where established approaches build directly on semantic models of machine code (foundational or policy specific), our approach builds on the semantics of higher level program descriptions, specifically CCL and C. Where established approaches are insensitive to compiler optimizations, our approach makes assumptions about the correspondence of assembly instructions to C programming statements. Certain optimizations (e.g., code reordering across the boundaries demarcated by calls to `__begin__` and `__inv__`) *may* break this correspondence. However, such optimizations have never been an issue during our experiments. In addition, the fundamental soundness theorem still holds because in the worst case, an optimization might result in a failure in proof checking but will never validate a proof for a program (optimized or otherwise) that violates a policy.

Eternal Limitations. Some limitations of our approach flow (seemingly inevitably) from the fundamental, and eternal, challenges of automated program verification. In particular, scalability is always a concern, and concurrency is usually a major source of exacerbation. Our approach assumes that components exhibit no internal concurrency. While the Copper model checker can effectively deal with concurrency, we know of no compact encoding proofs of concurrent programs. As mentioned earlier, small proof size is likely to be a major concern in the practical application of PCC.

Nevertheless, we believe that our work marks a positive and important step toward the development of rigorous, objective, and automated software-certification practices, and the reconciliation of formal and model-driven approaches for software development. Our experiment results are preliminary but realistic and encouraging, and therefore underline the need for further work in this direction.

⁷ For more information, go to <http://research.microsoft.com/TERMINATOR/default.htm>.

References

- [Appel 2001] Appel, A. W. “Foundational Proof-Carrying Code”. *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*. Boston, MA, June 16–19, 2001. Los Alamitos, CA: IEEE Computer Society, June 2001.
- [Ball 2001] Ball, T. & Rajamani, S. K. “Automatically Validating Temporal Safety Properties of Interfaces”, 103–122. Dwyer, M. B., editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of *Lecture Notes in Computer Science*. Toronto, Canada, May 19–20, 2001. New York, NY: Springer-Verlag, May 2001.
- [Bernard 2002] Bernard, A. & Lee, P. “Temporal Logic for Proof-Carrying Code”, 31–46. Voronkov, A., editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Computer Science*. Copenhagen, Denmark, July 27–30, 2002. New York, NY: Springer-Verlag, July 2002.
- [Chaki 2004a] Chaki, S.; Clarke, E.; Groce, A.; Jha, S.; & Veith, H. “Modular Verification of Software Components in C”. *IEEE Transactions on Software Engineering (TSE)* 30, 6 (June 2004): 388–402.
- [Chaki 2004b] Chaki, S.; Clarke, E. M.; Ouaknine, J.; Sharygina, N.; & Sinha, N. “State/Event-Based Software Model Checking”, 128–147. Boiten, E. A.; Derrick, J.; & Smith, G., editors, *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM '04)*, volume 2999 of *Lecture Notes in Computer Science*. Canterbury, UK, April 4–7, 2004. New York, NY: Springer-Verlag, April 2004.
- [Chaki 2005a] Chaki, S.; Ivers, J.; Sharygina, N.; & Wallnau, K. “The ComFoRT Reasoning Framework”, 164–169. Etessami, K. & Rajamani, S. K., editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, July 2005.
- [Chaki 2005b] Chaki, S. & Wallnau, K. *Results of SEI Independent Research and Development Projects and Report on Emerging Technologies and Technology Trends* (CMU/SEI-2005-TR-020, ADA449433). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. <http://www.sei.cmu.edu/publications/documents/05.reports/05tr020.html>.
- [Chaki 2006] Chaki, S. “SAT-Based Software Certification”, 151–166. Hermanns, H. & Palsberg, J., editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, volume 3920 of *Lecture Notes in Computer Science*. Vienna, Austria, March 25–April 2, 2006. Berlin, Germany: Springer-Verlag, March 2006.
- [Clarke 1982] Clarke, E. & Emerson, A. “Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic”, 52–71. Kozen, D., editor, *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture*

Notes in Computer Science. Yorktown Heights, New York, May 4–6, 1981. Berlin, Germany: Springer-Verlag, May 1982.

- [Hamid 2002]** Hamid, N. A.; Shao, Z.; Trifonov, V.; Monnier, S.; & Ni, Z. “A Syntactic Approach to Foundational Proof-Carrying Code”, 89–100. *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. Copenhagen, Denmark, July 22–25, 2002. Los Alamitos, CA: IEEE Computer Society, July 2002.
- [Henzinger 2002a]** Henzinger, T. A.; Jhala, R.; Majumdar, R.; Nacula, G. C.; Sutre, G.; & Weimer, W. “Temporal-Safety Proofs for Systems Code”, 526–538. Brinksma, E. & Larsen, K. G., editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*. Copenhagen, Denmark, July 27–31, 2002. New York, NY: Springer-Verlag, July 2002.
- [Henzinger 2002b]** Henzinger, T. A.; Jhala, R.; Majumdar, R.; & Sutre, G. “Lazy Abstraction”, 58–70. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, volume 37(1) of *SIGPLAN Notices*. Portland, OR, January 16–18, 2002. New York, NY: Association for Computing Machinery, January 2002.
- [Hissam 2005]** Hissam, S.; Ivers, J.; Plakosh, D.; & Wallnau, K. C. *Pin Component Technology (V1.0) and Its C Interface* (CMU/SEI-2005-TN-001, ADA441815). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.
<http://www.sei.cmu.edu/publications/documents/05.reports/05tn001.html>.
- [Ivers 2002]** Ivers, J.; Sinha, N.; & Wallnau, K. *A Basis for Composition Language CL* (CMU/SEI-2002-TN-026, ADA407797). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.
<http://www.sei.cmu.edu/publications/documents/02.reports/02tn026.html>.
- [Ivers 2004]** Ivers, J. & Sharygina, N. *Overview of ComFoRT: A Model Checking Reasoning Framework* (CMU/SEI-2004-TN-018, ADA442864). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004.
<http://www.sei.cmu.edu/publications/documents/04.reports/04tn018.html>.
- [Kupferman 2004]** Kupferman, O. & Vardi, M. Y. “From Complementation to Certification”, 591–606. Jensen, K. & Podelski, A., editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *Lecture Notes in Computer Science*. Barcelona, Spain, March 29–April 2, 2004. New York, NY: Springer-Verlag, March–April 2004.
- [Magee 2006]** Magee, J. & Kramer, J. *Concurrency: State Models and Java Programs*. Hoboken, NJ: John Wiley & Sons, 2006.
- [Namjoshi 2001]** Namjoshi, K. S. “Certifying Model Checkers”, 2–13. Berry, G.; Comon, H.; & Finkel, A., editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*. Paris, France, July 18–22, 2001. New York, NY: Springer-Verlag, July 2001.

- [Namjoshi 2003]** Namjoshi, K. S. “Lifting Temporal Proofs through Abstractions”, 174–188. Zuck, L. D.; Attie, P. C.; Cortesi, A.; & Mukhopadhyay, S., editors, *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '03)*, volume 2575 of *Lecture Notes in Computer Science*. New York, NY, January 9–11, 2002. New York, NY: Springer-Verlag, January 2003.
- [Necula 1996]** Necula, G. C. & Lee, P. “Safe Kernel Extensions Without Runtime Checking”, 229–243. *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI '96)*. Seattle, WA, October 28–31, 1996. New York, NY: Association for Computing Machinery, October 1996.
- [Necula 1997]** Necula, G. C. “Proof-Carrying Code”, 106–119. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. Paris, France, January 15–17, 1997. New York, NY: Association for Computing Machinery, January 1997.
- [Necula 1998a]** Necula, G. C. & Lee, P. “Efficient Representation and Validation of Proofs”, 93–104. *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS '98)*. Indianapolis, IN, June 21–24, 1998. Los Alamitos, CA: IEEE Computer Society, June 1998.
- [Necula 1998b]** Necula, G. C. & Lee, P. “Safe, Untrusted Agents Using Proof-Carrying Code”, 61–91. Vigna, G., editor, *Proceedings of Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. New York, NY: Springer-Verlag, 1998.
- [Necula 2001]** Necula, G. C. & Rahul, S. P. “Oracle-Based Checking of Untrusted Software”, 142–154. *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*, volume 36(3) of *SIGPLAN Notices*. London, UK, January 17–19, 2001. New York, NY: Association for Computing Machinery, January 2001.
- [Nelson 1980]** Nelson, G. “Techniques for Program Verification”. PhD diss., Stanford University, 1980.
- [Pnueli 1998]** Pnueli, A.; Siegel, M.; & Singerman, E. “Translation Validation”, 151–166. Steffen, B., editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *Lecture Notes in Computer Science*. Lisbon, Portugal, March 28–April 4, 1998. New York, NY: Springer-Verlag, March–April 1998.
- [Schneck 2002]** Schneck, R. R. & Necula, G. “A Gradual Approach to a More Trustworthy, Yet Scalable, Proof-Carrying Code”, 47–62. Voronkov, A., editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Computer Science*. Copenhagen, Denmark, July 27–30, 2002. New York, NY: Springer-Verlag, July 2002.
- [Wallnau 2003]** 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>.

- [Whalen 2002]** Whalen, M. W.; Schumann, J.; & Fischer, B. “Synthesizing Certified Code”, 431–450. Eriksson, L.-H. & Lindsay, P. A., editors, *Proceedings of the International Symposium on Formal Methods Europe (FME '02)*, volume 2391 of *Lecture Notes in Computer Science*. Copenhagen, Denmark, July 22–24, 2002. New York, NY: Springer-Verlag, July 2002.
- [Zhang 2003]** Zhang, L. & Malik, S. “Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications”, 10880–10885. *Proceedings of 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003)*. Munich, Germany, March 3–7, 2003. Los Alamitos, CA: IEEE Computer Society, March 2003.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE September 2007		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Certified Binaries for Software Components			5. FUNDING NUMBERS FA8721-05-C-0003	
6. AUTHOR(S) Sagar Chaki, James Ivers, Peter Lee, Kurt Wallnau, & Noam Zeilberger				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2007-TR-001	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2007-001	
11. SUPPLEMENTARY NOTES				
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12B DISTRIBUTION CODE	
13. ABSTRACT (MAXIMUM 200 WORDS) Proof-carrying code (PCC) and certifying model checking (CMC) are two established paradigms for obtaining objective confidence in the runtime behavior of a program. PCC enables the certification of low-level binary code against relatively simple (e.g., memory-safety) policies. In contrast, CMC provides a way to certify a richer class of temporal logic policies, but is typically restricted to high-level (e.g., source) code. In this report, an approach is presented to certify binary code against expressive policies, and thereby achieve the benefits of both PCC and CMC. This approach generates certified binaries from software specifications in an automated manner. The specification language uses a subset of UML statecharts to specify component behavior and is compiled to the Pin component technology. The overall approach thus demonstrates that formal certification technology is compatible with, and can indeed exploit, model-driven approaches to software development. Moreover, this approach allows the developer to trust the code that is produced without having to trust the tools that produced it. In this report details of this approach are presented and experimental results on a collection of benchmarks are described.				
14. SUBJECT TERMS certification, software validation, model checking, trust, safety, security			15. NUMBER OF PAGES 40	
16. PRICE CODE				
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	