Model-Driven Construction of Certified Binaries

Sagar Chaki* James Ivers* Peter Lee† Kurt Wallnau* Noam Zeilberger†

*Software Engineering Institute †Computer Science Department
Carnegie Mellon University

Abstract. Proof-Carrying Code (PCC) and Certifying Model Checking (CMC) are established paradigms for certifying the run-time behavior of programs. While PCC allows us to certify low-level binary code against relatively simple (e.g., memory-safety) policies, CMC enables the certification of a richer class of temporal logic policies, but is typically restricted to high-level (e.g., source) descriptions. In this paper, we present an automated approach to generate certified software component binaries from UML Statechart specifications. The proof certificates are constructed using information that is generated via CMC at the specification level and transformed, along with the component, to the binary level. Our technique combines the strengths of PCC and CMC, and demonstrates that formal certification technology is compatible with, and can indeed exploit, model-driven approaches to software development. We describe an implementation of our approach that targets the Pin component technology, and present experimental results on a collection of benchmarks.

1 Introduction

Today, off-the-shelf programs are increasingly 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 distributed in executable machine code, or "binary" form. In this article we present a framework for generating trustworthy "binaries" from component specifications, and for proving that such binaries satisfy specific policies. A more detailed exposition of this work is available as a technical report [1].

Our approach builds on two existing paradigms for software certification: proof-carrying code and certifying model checking. Proof-Carrying Code (PCC) [2] constructs a proof that machine code respects a desired policy, packages the proof with the code so that the validity of the proof and its relation to the code can be independently verified before the code is deployed. In contrast, Certifying Model Checking (CMC) [3] is an extension of model checking [4] for generating "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 [5, 6].

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[1] policies. The progress of PCC has also been hindered by the need for manual intervention, e.g., to specify loop invariants. In contrast, CMC is able to certify programs against a richer class of temporal logic policies (which subsumes both safety and liveness), and is automated. However, CMC is only able to certify 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 PCC in resource constrained environments. Unfortunately, embedded software (e.g., in medical devices) that might benefit from the high confidence obtained with PCC are almost certainly going to 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 subsumes both safety and liveness. We use the state/event-based temporal logic called SE-LTL [7] developed at the SEI.
3. **Automation:** As with CMC, we employ iterative refinement in combination with predicate abstraction and model checking to generate appropriate invariants and 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.

2 Basic Concepts

In this section, we present the basic concepts of components, policies, ranking functions, verification conditions, certificates, etc., that we use later.

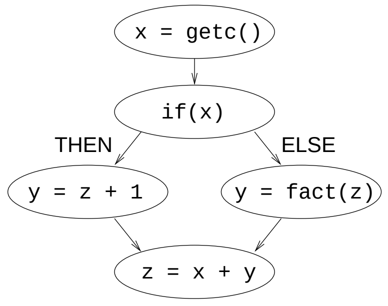
Logical Foundation. We assume a denumerable set of variables *Var*, and a set of expressions *Expr* constructed using *Var* and the standard C operators. We view every expression as a formula in quantifier-free first order logic with C interpretations for operators and truth values (0 is false and anything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

Component. We deal with several forms of a component—their Construction and Composition Language (CCL) form, C implementation form, analysis form,

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

and their binary (assembly language) form. The syntax and semantics of CCL have been presented elsewhere [8], and we use the PowerPC assembly language. Hence, we only describe the other two (analysis and C implementation) forms.

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.



```
x = getc();
if(x) y = z + 1;
else y = fact(z);
z = x + y;
```

Fig. 1. Component in Analysis (Left) and C (Right) Forms.

The C implementation is generated from CCL, and contains both the logical behavior specified by Statecharts, and the infrastructure imposed by the Pin [9] component model. 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) are possible. 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 paper, 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 [7], and are not crucial to grasp 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 some objective O. 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 O. Then, proving liveness boils down to proving that with every transition, the rank of the current system state decreases appropriately, i.e., the system makes progress toward O. Since there are no infinite decreasing chains in R, the system must eventually attain O. 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. Given any ranking function RF, C and φ are known implicitly.*

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

Ranking functions, verification conditions, and software certification are related intimately, as expressed in 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 have been presented elsewhere [2, 6].

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 resolution proof of the validity of $VC(RF)$.*

Indeed, if such a certificate (RF, Π) exists, then, by the soundness of resolution[2], 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: (i) compute $VC(RF)$ using VC-Gen, and (ii) verify that Π is a correct proof of $VC(RF)$ using a proof checker.

[2] More details on resolution can be found in our technical report [1].

– 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 [2]. This fact is not altered even if the binary program, the proof certificate, or both, are tampered with. 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 when incorporating components derived from unknown and untrusted sources.

3 Framework for Generating Certified Binaries

Figure 2 depicts our infrastructure for certified component binary generation. Key elements are numbered for each 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. Certified component binaries are generated step-wise as follows:

Step 1. A component is specified in CCL [8]. 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 that the component is to be certified against.

Step 2. *Spec* is transformed ("interpreted" [10]) into a component C, that can be processed by a model checker. C is comprised of a C program along with finite state machine specifications for procedures invoked by the program. This step was implemented by augmenting prior work [11] so that C contains additional information relating its line numbers, variables and other data structures with those of *Spec*. This information is crucial for the subsequent reverse-interpretation of ranking functions in Step 4.

Step 3. C is input to Copper, a state-of-the-art certifying software model checker that 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.

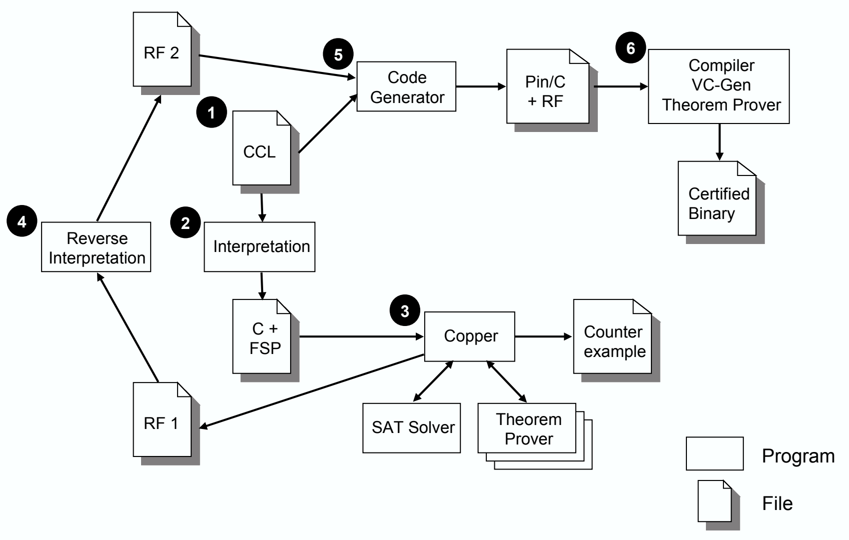


Fig. 2. Framework for Generating Certified Binaries.

Step 4. The certificate *RF1* only certifies C (the result of the interpretation) against the policy φ. It is reverse-interpreted into a certificate $\mathit{RF2} : \mathit{Expr} \rightarrow R$ such that $VC(\mathit{RF2})$ is valid. This process is enabled by the additional information generated during interpretation to connect *Spec* with C in 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 [9]. We augmented an existing Pin/C code generator to also create a ranking function, using *RF2*, and embed it in the generated code. In essence, we transform the ranking function, and the component, from CCL to the Pin/C formalism.

Step 6. The final step consists of three distinct sub-steps.

Step 6.1. The component with the embedded ranking function is compiled from Pin/C to binary form. In our implementation we use GCC[3] (targeting the PowerPC instruction set) for this step. Let *RF3* be the ranking function embedded in the binary obtained as a result.

Step 6.2. We compute $VC(\mathit{RF3})$ using VC-Gen.

Step 6.3. We obtain a proof Π of $VC(\mathit{RF3})$ using a proof-generating theorem prover. In our implementation we use a SAT-based theorem prover for this step. In essence, we convert $\neg VC(\mathit{RF3})$ (i.e., the logical negation of $VC(\mathit{RF3})$) to a Boolean formula ϕ. We then check if ϕ is unsatisfiable using ZChaff [12]. If $\neg VC(\mathit{RF3})$ is unsatisfiable, i.e., if $VC(\mathit{RF3})$ is valid, then the resolution proof emitted by ZChaff serves as Π. The use of SAT enables us to obtain extremely compact proofs [6] in practice. Finally, the certificate (RF, Π) along with the binary is emitted as the end result—the certified binary for *Spec*.

[3] `http://gcc.gnu.org`

Trusted Computing Base. It is instructive to discuss the artifacts that must be trusted for our approach to be effective. In essence, the TCB is comprised of: (1) VC-Gen, (2) the procedure for converting $\neg\, VC(RF3)$ to ϕ, and (3) the procedure for checking that Π refutes ϕ. All of these procedures are computationally inexpensive and can be implemented by small programs. Thus, they are more trustworthy (and more verifiable) than the rest of the programs of Figure 2. Note that the interpreter, the certifying model checker, the reverse-interpreter, the code generator, the compiler, and the theorem prover are not in the TCB. Each of these tools is quite complex, and their elimination from the TCB raises considerably the degree of confidence of our certification method.

How the TCB is demarcated and how its size and complexity is reduced is an important theoretical and practical concern for future applications of PCC. There are several approaches to this concern. For example, "foundational" PCC [13] 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 [14]. Even this more "pragmatic" approach can achieve good results. In our own implementation, the TCB is over fifteen times smaller in size (30 KB vs. 450 KB) than the rest of the infrastructure.

4 Certifying Model Checking

The infrastructure for performing certifying model checking corresponds to steps 1-4 from Figure 2. We begin with component specifications expressed in CCL.

Overview of CCL. CCL is a simple composition language for describing component behavior and how components are wired together into assemblies for deployment. In CCL, a component is viewed as a collection of potentially concurrent units of computation called *reactions*, each of which describes how the component responds to stimuli on its sink pins and under what circumstances it initiates interactions on its source pins. Figure 3 shows a CCL specification for a component `comp` with a single reaction `R`. The reaction `R` reacts to stimuli from its environment on its `incr` sink pin by incrementing an internal counter (up to a maximum and then reseting to a minimum) and informing its environment of the new value on its `value` source pin. 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 differs from Statecharts in two important ways:

1. CCL does not permit some concepts defined in UML Statecharts, most significantly hierarchical states and concurrent sub-states (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.

```
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
```

Fig. 3. CCL Specification for a Simple Component.

Interpreting CCL to C. CCL specifications are transformed into an equivalent representation in C and FSP [15] for use with Copper, a software model checker. This corresponds to Step 2 from Figure 2. 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 FSP. 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 ^`Pin` for initiation on a pin `Pin`, or a $`Pin` for completion on a pin `Pin`. 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. Thus, when a component is willing to engage in an interaction over any of several sink pins (i.e., pull the next message from its queue and respond accordingly), this corresponds to a willingness to synchronize over one of several FSP-style events. This concept is not as easily represented in C, and we use Copper's ability to provide specifications of functions. We insert a call to an `fsp_exernalChoice`() 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 the node in the CCL abstract syntax tree (AST) of the CCL specification that corresponds to C statement that follows the annotation. These calls are known to Copper, and are 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 emitted upon successful verification.

Ranking Function Generation by Copper. Copper uses iterative-predicate-abstraction-refinement for verification. This paradigm has been presented in detail elsewhere [16–18, 1] and we only present its relevant features here. The key idea is that conservative models of the C program are constructed via predicate abstraction, verified, and refined iteratively until either the verification succeeds, or a real counterexample is found. Let M be the model verified successfully. Then 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. Each valuation V has a concretization $\gamma(V) \in \mathit{Expr}$. Also, for any two distinct valuations V and V', $\gamma(V)$ and $\gamma(V')$ are logically disjoint.

We now describe the ranking function generated by Copper. The ranking function is generated as a set of triples of the form $((l, I), s, r)$ where: (i) I is an invariant, i.e., the concretization of a predicate valuation V such that (l, V) is a reachable state of M[4], (ii) s is a state of the Büchi automaton corresponding to the policy, and (iii) r is a rank. The procedure for constructing an appropriate ranking function is presented elsewhere [6] and we do not describe it further.

Recall, from Definition 1, that a ranking function is a mapping from expressions to ranks. Each triple $((l, I), s, r)$ emitted by Copper corresponds to an entry in this mapping as follows. Let PC and SS be special variables representing the program location (i.e., program counter) and the policy 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 $((l', I'), s', r')$ emitted 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 emitted is always well-formed.

Figure 4 shows an excerpt from the ranking function generated for our example CCL specification and a policy asserting that `min` <= `i` <= `max` is always true (RF 1 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.

[4] 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.

```
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 )]
```

Fig. 4. Ranking Function in terms of Interpreted C Program.

The final step in certifying model checking is to relate the ranking function back to the original CCL specification. This 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 and predicates relating to variables that were introduced during interpretation are stripped or remapped to the appropriate CCL concepts. At the conclusion of certifying model checking, if a component is known to satisfy all of its policies, we obtain evidence to that effect in the form of a ranking function expressed in terms of nodes of the AST 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 nodes of its AST. From previous work, we have a code generator for CCL that generates C code targeted for deployment in the Pin component technology (Pin/C). 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 to maintain a correlation between the location of these invariants in the Pin/C code and the assembly code resulting from compilation.

The convention we chose (shown in Figure 5) encodes invariants using a pair of function calls inserted 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.

We extended the Pin/C code generator to insert these pairs of calls at any locations for which the ranking function provides invariants (a short excerpt from the generated code is shown in Figure 6). The code generator also adds an additional predicate to each invariant in the ranking function, an encoding of the current state of the state machine. 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 is crucial for effective computation of the certified binary, as presented in the next section, without having to supply loop invariants.

```
__begin__();                              bl __begin__
                                          li %r0,0
                                          stw %r0,16(%r31)
                                          lwz %r0,8(%r31)
                                          cmpwi %cr7,%r0,0
                                          blt %cr7,.L5
                                          lwz %r0,8(%r31)
                                          cmpwi %cr7,%r0,9
                                          bgt %cr7,.L5
                                          li %r0,1
                                          stw %r0,16(%r31)
                                      .L5:
                                          lwz %r3,16(%r31)
                                          crxor 6,6,6
__inv__((n > =0) && (n < 10));            bl __inv__
```

Fig. 5. Invariants in Pin/C Code (Left) and Assembly Code (Right).

```
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 */ ) {
      ...
```

Fig. 6. Excerpt from Generated Pin/C Code.

6 Certified Binary Generation

In this section we describe the process of obtaining the end-product of our approach, 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 which is essentially the proof of a verification condition.

The certified binary is obtained by simply compiling this C source code with any standard compiler. In our implementation, we used GCC targeted at the PowerPC instruction set for this step of our procedure. The binary generated by the compiler contains assembly instructions, peppered with calls to `__begin__()` and `__inv__(...)`. 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 compute and store the value of the argument being passed to `__inv__(...)` in register `r3`.

To construct the certificate, we first construct the verification condition VC. This 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, viz., 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. We leverage our previous work on using a theorem-prover based on Boolean satisfiability (SAT) [6] to generate extremely compact certificates as 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.

Given a binary B and an associate certificate C, we validate B as follows. 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 indeed a proper proof of VC.

Note that once a certified binary has been validated successfully, the embedded binary invariants are stripped off before the binary is actually deployed. This is crucial for both correctness (since what we really certify is the binary *without* the invariants) and performance. Finally, it is noteworthy that our choice of mechanism for carrying invariants from C code to assembly code is sensitive to compiler optimizations. Certain optimizations (e.g., code reordering across the boundaries demarcated by calls to `__begin__` and `__inv__`) *may* break this correspondence. Fortunately, the fundamental soundness theorem still holds. In the worst case, such a reordering might result in a failure in proof checking, but will never validate a proof for a program that violates a policy.

7 Related Work

PCC was proposed by Necula and Lee [19, 2, 20] for certifying memory safety policies on binaries. PCC works by hard-coding the desired safety policies within the machine instruction semantics, while our approach works at the specification level and encodes the policy as a separate automaton. Foundational PCC [13, 21] attempts to reduce the trusted computing base of PCC to include only the foundations of mathematical logic. Bernard and Lee [22] propose a new temporal logic to express PCC policies for machine code. Non-SAT-based techniques for minimizing PCC proof sizes [23, 24] have also been proposed. Whalen et al. [25] describe a technique for synthesizing certified code. They augment the AUTO-

BAYES 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 [3, 26] emit an independently checkable certificate of correctness when a temporal logic formula is found to be satisfiable by a finite state model. Namjoshi [27] has proposed a two-step technique for obtaining proofs of Mu-Calculus policies on infinite state systems. 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 [16], BLAST [17] and MAGIC [18]. While SLAM and MAGIC do not generate any proof certificates, BLAST implements a method [5] for lifting proofs of correctness. 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 (which is difficult and yet to be widely adopted) or via translation validation [28] (which still assumes that the source code is correct). In contrast, our approach requires no such correctness assumptions.

In previous work, we developed an expressive linear temporal logic called SE-LTL [7] that can be used to express both safety and liveness claims of component-based software. In the work reported here, we modified 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 [29]. 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 [6]. 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 simple CCL specification of a component that manipulates an integer variable and the 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 5 minutes with modest memory requirements.

To validate the translation of a certified C component to a certified binary (Step 6 in Figure 2), we conducted additional experiments with Micro-C, a lightweight operating system for embedded real-time applications. The OS source code consists of about 6000 lines of C (97 KB) and uses a semaphore to ensure mutually exclusive access to shared kernel data structures. Using out approach we were able to certify that all kernel routines follow the proper locking order when using the semaphore. 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, we certified, using our approach, 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 show that our approach is viable, we believe that a more robust implementation and more realistic case studies are needed in order to push our technique amongst a wider user base.

9 Conclusion

In this paper, we presented an automated approach for generating certified binaries from software component specifications. Our technique 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 addition, we developed and experimented with a prototypical implementation of our technique. Our implementation, and our overall approach, does have limitations which we like to classify into the following two broad categories:
Deferred Features. Some of the missing features from our implementation are not difficult conceptually, but are best deferred until a target environment 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. CCL currently supports only a primitive assortment of types, and, as a consequence, 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). We have also 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 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 is able to certify a restricted set of programs. More general ranking functions are generated by other tools such as Terminator[5].

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.

[5] `http://research.microsoft.com/TERMINATOR/default.htm`

References

1. Chaki, S., Ivers, J., Lee, P., Wallnau, K., Zeilberger, N.: ”certified binaries for software components”. Technical Report CMU/SEI-2007-TR-001 (2007)
2. Necula, G., Lee, P.: Safe Kernel Extensions without Runtime Checking. In: OSDI’96
3. Namjoshi, K.S.: Certifying Model Checkers. In: Proceedings of the 13th International Conference on Computer Aided Verification (CAV ’01). (2001)
4. Clarke, E., Emerson, A.: Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In: Proc. of WLP. (1982)
5. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-Safety Proofs for Systems Code. In: Proc. of CAV. (2002)
6. Chaki, S.: SAT-Based Software Certification. In: Proc. of TACAS. (2006)
7. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-Based Software Model Checking. In: Proc. of IFM. (2004)
8. Wallnau, K., Ivers, J.: ”Snapshot of CCL: A language for predictable assembly”. Technical note CMU/SEI-2003-TN-025, Software Engineering Institute (2003)
9. Hissam, S., Ivers, J., Plakosh, D., Wallnau, K.C.: ”Pin Component Technology (V1.0) and Its C Interface”. Technical Report CMU/SEI-2005-TN-001 (2005)
10. Ivers, J., Sinha, N., Wallnau, K.: ”A Basis for Composition Language CL”. Technical Report CMU/SEI-2002-TN-026, Software Engineering Institute (2002)
11. Ivers, J., Sharygina, N.: ”Overview of ComFoRT: A Model Checking Reasoning Framework”. Technical Report CMU/SEI-2004-TN-018 (2004)
12. Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In: DATE’03
13. Appel, A.W.: Foundational proof-carrying code. In: Proc. of LICS. (2001)
14. Schneck, R.R., Necula, G.: A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In: Proc. of CADE. (2002)
15. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. (2006)
16. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: Proc. of SPIN. (2001)
17. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: POPL’02
18. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Transactions on Software Engineering (TSE) (6) (2004)
19. Necula, G.C.: Proof-Carrying Code. In: Proc. of POPL. (1997)
20. Necula, G.C., Lee, P.: Safe, Untrusted Agents Using Proof-Carrying Code. In: Proceedings of Mobile Agents and Security. (1998)
21. Hamid, N.A., Shao, Z., Trifonov, V., Monnier, S., Ni, Z.: A Syntactic Approach to Foundational Proof-Carrying Code. In: Proc. of LICS. (2002)
22. Bernard, A., Lee, P.: Temporal Logic for Proof-Carrying Code. In: CADE. (2002)
23. Necula, G., Lee, P.: Efficient Representation and Validation of Proofs. In: LICS’98
24. Necula, G., Rahul, S.: Oracle-Based Checking of Untrusted Software. In: POPL’01
25. Whalen, M.W., Schumann, J., Fischer, B.: Synthesizing certified code. In: Proc. of FME. (2002)
26. Kupferman, O., Vardi, M.: From Complementation to Certification. In: TACAS’04
27. Namjoshi, K.S.: Lifting Temporal Proofs through Abstractions. In: VMCAI. (2003)
28. Pnueli, A., Siegel, M., Singerman, E.: ”translation validation”. In: TACAS. (1998)
29. Chaki, S., Wallnau, K.: ”Results of SEI Independent Research and Development Projects and Report on Emerging Technologies and Technology Trends”. Technical report CMU/SEI-2005-TR-020, Software Engineering Institute (2005) Chapter 6.