Overview of ComFoRT: A Model Checking Reasoning Framework

James Ivers
Natasha Sharygina

April 2004

Predictable Assembly from Certifiable Components Initiative

Technical Note
CMU/SEI-2004-TN-018

This work is sponsored by the U.S. Department of Defense.

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

This work was created in the performance of Federal Government Contract Number F19628-00-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).

Contents

List of Figures

Acknowledgements

The ComFoRT reasoning framework and this report have benefited from the contributions of many people. Professor Edmund Clarke at Carnegie Mellon University and his model checking group have been valuable collaborators in our application of model checking for component-based software verification. Sagar Chaki developed the MAGIC model checker from which the model checking engine in ComFoRT was derived. Sagar Chaki, Joel Ouaknine, and Nishant Sinha helped us develop new model checking techniques used in ComFoRT that are specifically designed to verify component-based software designs. Members of the Predictable Assembly from Certifiable Components (PACC) Initiative developed and refined the prediction-enabled component technology (PECT) concept, including reasoning frameworks as a means of packaging complex analyses. Kurt Wallnau helped design the ComFoRT interpretation and provided us with an initial implementation. Finally, Scott Hissam, Linda Northrop, and Kurt Wallnau provided thoughtful reviews that greatly improved the quality of this report.

Abstract

Component technologies are gaining acceptance in the software community as effective tools to quickly assemble increasingly complex systems from components. Most of the current component technologies, however, fail to help developers predict important software qualities like performance, safety, and reliability. A prediction-enabled component technology (PECT) augments the capabilities of a component technology with one or more reasoning frameworks that package quality-specific analyses and the means to apply them to component-based systems. Model checking is an automated approach for exhaustively analyzing whether systems satisfy specific behavioral claims that can be used to characterize safety and reliability requirements. This technical note describes ComFoRT, a reasoning framework that packages the effectiveness of state-of-the-art model checking in a form that enables users to apply the analysis technique without being experts in its use, and its incorporation in a PECT.

1 Introduction

Across the software industry, projects are challenged to produce software that meets ubiquitous demands for increased functionality, better quality, and reduced time to market while simultaneously satisfying performance, safety, or security requirements. While the use of component technologies addresses the first set of demands, current processes and technologies fail to help developers predict the qualities of a system of components, resulting in expensive integration and testing efforts.

Early architectural decisions have a large impact on software qualities [Bass 03] but are rarely analyzed by component technology tools, which are better suited to traditional, syntactic integration problems like those checked by compilers and linkers. Though theories and tools for analyzing software qualities exist, many require theory-specific expertise to use effectively, and few (if any) are included in the suite of tools accompanying most component technologies.

The Predictable Assembly from Certifiable Components (PACC) Initiative at the Carnegie Mellon® Software Engineering Institute develops technologies and methods for bringing the benefits of quality-specific analyses to component technologies. Doing so provides a means to reliably predict the runtime qualities (e.g., performance or reliability) of assemblies of components from their certifiable properties (e.g., execution time or behavioral descriptions). Such predictions augment the quick-assembly capability of component technologies with an ability to determine whether a particular design will satisfy its quality requirements.

One of the analysis techniques we are applying is formal verification by model checking. Model checking is an automated algorithmic approach for exhaustively analyzing whether concurrent finite-state models satisfy specific behavioral claims. The types of claims evaluated are temporal expressions over system execution—for example, checking whether a system can ever fail to answer a message while in a normal mode of operation or deadlock. Checking these types of claims allows developers to determine whether systems will respond correctly and satisfy specific safety and reliability requirements.

While model checking has been successfully applied to software, model checking successes are far more common in hardware industry applications and in research settings than in industrial software development. Two reasons for the dearth of software successes are the theoretical problems limiting successful application to software (e.g., state space explosion) and the fact that commercial developers typically lack the expertise needed to apply model

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

checking. Our approach to addressing these problems is to incorporate state-of-the-art model checking into a prediction-enabled component technology (PECT).

A PECT packages the complexities of an analysis technology (e.g., model checking) in a reasoning framework that is combined with a component technology in a way that allows developers to predict the behavior of their component-based systems without having to become experts in the analysis technology. Reasoning frameworks also exploit a component technology's features and constraints to scope the class of designs that must be considered, which can alleviate theoretical problems and improve applicability. In this document, we describe how a particular model checking approach is incorporated into a PECT by the creation of the model checking reasoning framework ComFoRT (Component Formal Reasoning Technology).

Section 2 describes how a PECT integrates a component technology and a reasoning framework to ensure that component-based designs will be predictable with respect to one or more qualities of interest.

Section 3 provides more information on model checking, the types of claims it can analyze, and the current state of the research and practice of model checking.

Section 4 describes ComFoRT, the model checking reasoning framework that we are developing, and some of the technical challenges involved.

Section 5 summarizes the current state of ComFoRT and discusses next steps.

2 Prediction-Enabled Component Technology (PECT)

As shown in Figure 1, a PECT extends the notion of a component technology with one or more reasoning frameworks providing the analyses needed to predict specific runtime qualities of assemblies of components (or simply "assemblies") [Wallnau 03a].

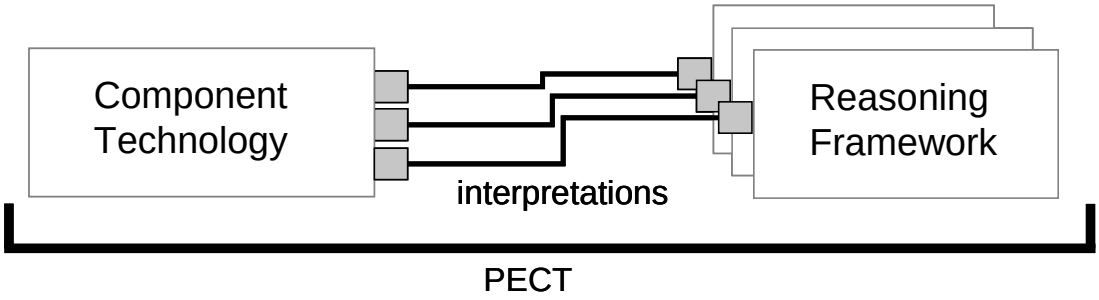

Figure 1: Simple View of a PECT

A component technology consists of a component model and a runtime environment [Bachmann 00]. The component model specifies allowable component types, interaction mechanisms, and services, and constrains how these features can be used together. A runtime environment is an execution environment that enforces aspects of the component model. A runtime environment plays a role analogous to that of an operating system—serving as the context in which components execute and providing implementations of interaction mechanisms (e.g., synchronous and asynchronous communication) and services.

Each reasoning framework embodies the concepts and theories needed to analyze, and hence predict, specific emergent qualities of an assembly. In practice, different reasoning frameworks are needed to predict different runtime qualities (e.g., performance, reliability, security, or safety), use different component properties[1] as inputs, and impose different constraints on what constitutes an analyzable design.

The mechanics of extending a component technology with a reasoning framework are a little more complicated than shown in Figure 1. Since the goal is to predict the behavior of assemblies, not the component technology or its runtime, the box labeled "Component Technology" must be understood to represent the set of assemblies that can be constructed using the component technology.

1. A property is a descriptive characteristic of an element (e.g., the priority of a thread is a property of the thread). Syntactically, a property has a name and a value.

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

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

Each reasoning framework is applied to assembly specifications by means of a formal interpretation that generates reasoning framework specific models from CCL specifications. While a particular interpretation often requires reasoning framework specific information (e.g., execution time or thread priority) that is captured in CCL as property annotations, much of the information in a CCL specification (e.g., topological information and the choice of interaction mechanisms) is used by most reasoning frameworks' interpretations.

Packaging analysis techniques in reasoning frameworks that include an automated interpretation is primarily a way of packaging complexity and expertise. The intent is to allow PECT users (software architects or developers) to gain the benefit of state-of-the-art analyses without having to become experts in the underlying theories and tools.

PECT users design components and assemblies using CCL as their design language and add property annotations as needed to enable specific predictions (i.e., to satisfy specific reasoning framework constraints). PECT automation handles the rest: it generates reasoning framework specific models by interpretation, uses a reasoning framework to compute predicted behavior, and translates the results back into concepts from the original CCL specifications.

3 Model Checking

It is becoming increasingly important that software systems be more robust and reliable. As the complexity of such systems grows, conventional testing methods are increasingly inadequate to ensure reliability because testing cannot practically achieve complete coverage. A more complete approach, which is gaining acceptance in industry, is to use formal methods to reason about system correctness (witness the active projects at AMD [Russinoff 98], Cadence [Barakatain 01], IBM [Ben-David 03, Ziv 03, IBM 04], Intel [Gerth 01], Lucent [Godefroid 97, Chandra 02], Microsoft [Ball 04], Motorola [Abadir 03], NASA [Havelund 00, Nelson 03, Lindsey 04], National Semiconductors Corp. [0-In 03], etc.).

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

Model checking is a technique for verifying finite-state concurrent systems.[2] One benefit of restricting ourselves to finite-state systems is that verification can be performed automatically. Given sufficient resources, model checking always terminates with a *yes* or *no* answer. Moreover, it can be implemented by algorithms that have reasonable efficiency and that can be run on moderate-sized machines.

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

2. The word *system* is used to refer to an artifact that describes the behavior of a software or hardware system. The kinds of artifacts typically verified using model checking include software designs (e.g., Unified Modeling Language (UML) statecharts), software implementations (e.g., C or Java source code), and hardware designs (e.g., Verilog code).

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

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

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

3.1 The Process of Model Checking

To model check a system, the following steps are performed:

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

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

3.2 Software Model Checking Challenges

Model checking is efficient in hardware verification, but applying it to software is complicated by several factors, ranging from the difficulty of modeling computer systems—due to the complexity of programming languages as compared to hardware description languages—to difficulties in specifying meaningful claims for software using the usual temporal logical formalisms of model checking. The most significant limitation, however, is

the *state space explosion* problem (which applies to both hardware and software), whereby the complexity of model checking becomes prohibitive.

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

Another significant limitation in model checking software stems from the *limited expressiveness* of classical temporal logics. When verifying concurrent software, one needs to specify both *state* information (e.g., program counter location or memory contents) and *communication* among concurrent units. For example, the Bluetooth L2CAP specification[3] asserts that "when an `L2CAP_ConnectRsp` event is received in a `W4_L2CAP_CONNECT_RSP` state, within one time unit, an L2CAP process may send out an `L2CAP_ConnectInd` event, disable the RTX timer, and move to state `CONFIG`." As this example shows, both states (`W4_L2CAP_CONNECT_RSP` and `CONFIG`) and events (`L2CAP_ConnectRsp` and `L2CAP_ConnectInd`) are required to properly capture the desired L2CAP behavior.

Generally, in concurrent software, communication among concurrency units occurs via actions (events) that can represent function calls, requests, acknowledgments, and so forth. These communications can be data dependent and carry data on their channels. Existing model checking techniques typically use either *state-based* or *event-based* formalisms to represent finite-state models of software. In principle, the frameworks are interchangeable: an event can be encoded as a change in state variables, or different events can be used with a state to reflect different values of the software's internal variables. Neither approach on its own is practical, however, when it comes to the specification of data-dependent communication claims: considerable domain expertise is then required to annotate the model and to specify proper specifications in a temporal logic.

3.3 Current Research in Software Model Checking

The ComFoRT reasoning framework exploits current research efforts in the model checking community. Specifically, it focuses on resolving software verification limitations outlined in the previous section.

Ameliorating state space explosion is the major research problem. There are three main approaches to handling this problem:

3. Haartsen, J. *Bluetooth Baseband Specification, Version 1.0*. Published in 2003.

1. **Compositional reasoning**. Verification is partitioned into checks of individual modules while the global correctness of the composed system is established by constructing a proof outline that exploits the modular structure of the system.

2. **Abstraction**. A smaller abstract system is constructed such that the claim holds for the original system if it holds for the abstract system.

3. **Counterexample-guided abstraction refinement**. Abstracted systems are refined iteratively using information extracted from counterexamples until an error is found or it is proven that the system satisfies the verification claim.

3.3.1 Compositional Reasoning

Since model checking was created for verifying hardware systems and since most hardware designs have a natural division into modules, one extension of model checking to larger designs was accomplished by taking a "divide and conquer" approach. Under the approach, a verification claim for a system is decomposed into a set of local claims for the system modules, and then each is verified separately [Clarke 92].

The compositional approach establishes whether a system composed of modules M_1, M_2 satisfies a claim φ (written $M_1 \| M_2 \models \varphi$). A naïve compositional approach executes (1) $M_1 \models \varphi$ and (2) $M_2 \models \varphi$ and concludes by proving that (3) $M_1 \| M_2 \models \varphi$. Unfortunately, this naïve approach is not sound when both M_1 and M_2 satisfy φ only in a suitable constraining environment. To solve this problem, the compositional principle is usually strengthened to an *assume-guarantee principle*: it executes (1) $M_1 \| \varphi_2 \models \varphi_1$ and (2) $M_2 \| \varphi_1 \models \varphi_2$ and concludes by proving that (3) $M \models \varphi$ [Abadi 95, Alur 96, Clarke 89, Kurshan 94, McMillan 97, McMillan 98, Misra 81, Pnueli 85, Stark 85]. This obligation uses the local claims φ_1, φ_2 as the constraining environments (assumptions) with regard to the behavior of M_2, M_1 taken in isolation from M_1, M_2 respectively.

In general, for a system composed of multiple modules, assume-guarantee reasoning succeeds as long as it can be shown that each system module M_i satisfies a corresponding local claim φ_i under a suitable constraining environment. The assume-guarantee reasoning approach has been successful in verifying large hardware systems, but there are some major difficulties in its application to software systems, most notably in (1) decomposing the system and (2) identifying suitable environment assumptions. Moreover, in some cases the complex dependencies among modules make it impossible to decompose claims into local claims of modules.

3.3.2 Abstraction

Abstraction is one of the principal complexity reduction techniques [Ball 01a, Clarke 92, Dams 94, Kesten 00, Kurshan 94, Loiseaux 95]. Abstraction techniques reduce the state

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

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

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

3.3.3 Counterexample-Guided Abstraction Refinement

Effective model checking of realistic systems generally requires a combination of various state space reduction techniques. One of the most promising is *counterexample-guided abstraction refinement* (CEGAR) [Kurshan 94]. It uses automated abstraction procedures and has been used successfully to verify industrial hardware [Clarke 00] and software [Ball 01b] systems.

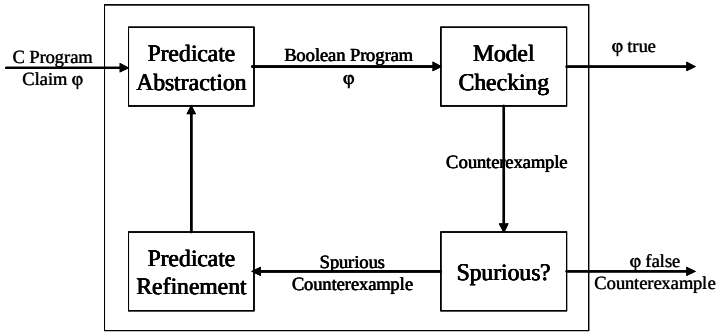

Figure 2: The Counterexample-Guided Abstraction Refinement Framework

The CEGAR framework is shown in Figure 2. It iteratively computes more and more precise abstractions of a system (a C program) until a valid counterexample is found or the claim is found to be correct. Initially, a very coarse but conservative abstraction is generated. The conservative abstraction ensures that if model checking shows that the claim holds for the abstract system, it also holds on the original system, at which point model checking terminates. If the claim does not hold for the abstract system, the model checker provides an abstract counterexample. This counterexample is then checked against the original, concrete system. If the check succeeds, the counterexample is valid, and the claim is false. If not, the counterexample is spurious and used to refine the abstraction, and the process starts over.

4 The ComFoRT Reasoning Framework

The objective of the ComFoRT reasoning framework is twofold: (1) to use model checking to predict whether assemblies will meet specific safety and reliability requirements and (2) to allow developers to apply model checking without having to become experts in model checking theory or tools. To achieve these goals, we developed a model checking engine for ComFoRT that uses state-of-the-art model checking algorithms for software verification. The model checking engine is derived from MAGIC,[4] a tool developed by the model checking group at Carnegie Mellon University (CMU).

Figure 3 shows one effective way to use ComFoRT.

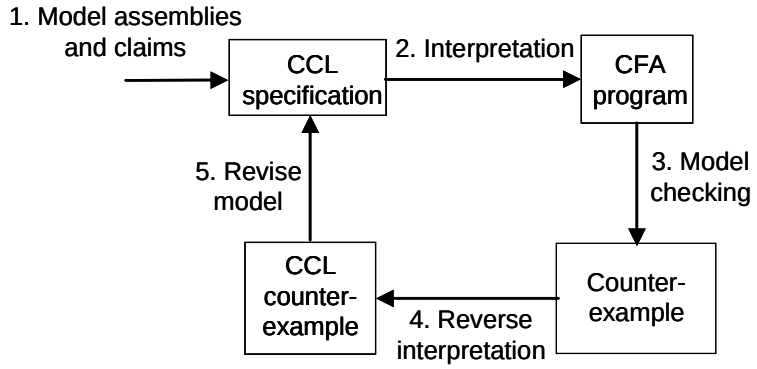

Figure 3: Applying the ComFoRT Reasoning Framework

1. PECT users model their components and assemblies in CCL and specify the claims to be checked.
2. An automated interpretation generates an input program for the model checking engine from the CCL specifications.
3. The model checking engine checks whether the provided models satisfy the specified claims and generates counterexamples for those claims not satisfied.
4. An automated reverse interpretation expresses the model checking results in terms of the CCL specifications, where the results (predictions) are recorded as property annotations.

PECT users evaluate the predictions to decide how the design needs to be changed to satisfy any unsatisfied claims. The following sections describe how the first four steps are supported by ComFoRT. Section 4.1 describes the model checking engine (used in Step 3). Sections 4.2

4. Chaki, S.; Clarke, E.; Groce, A.; Ouaknine, J.; Strichman, D.; & Yorav, K. "Efficient Verification of Sequential and Concurrent C Programs." To be published in the journal *Formal Methods in System Design*.

to 4.4 describe the tools and concepts that are used to package the model checking engine in a reasoning framework and that are applied in the remaining steps (1, 2, and 4).

4.1 Model Checking Engine

An important goal of ComFoRT is to achieve scalable verification of component-based software systems. Consequently, support for the time-proven techniques of abstraction and compositional reasoning—key factors in scaling model checking to software verification—guided our development of a model checking engine. We were able to use the state-of-the-art software verification tool MAGIC as a starting point because of its support for these techniques.

ComFoRT uses automated predicate abstraction techniques from MAGIC to create finite-state models of software. Counterexample validation and abstraction refinement procedures from MAGIC are used within a fully automated CEGAR loop to reduce verification complexity. These techniques are elaborated on in Sections 4.1.1 and 4.1.2. The model checking engine in ComFoRT provides new verification techniques described in Sections 4.1.3 and 4.1.4 that, when used in combination with MAGIC abstractions and CEGAR algorithms, improve the model checking suitability for analyzing component-based software designs.

The input to the ComFoRT model checking engine is a program expressed in combinations of C code, FSP expressions,[5] and auxiliary statements. For simplicity, we refer to such programs as CFA (C, FSP and auxiliary) programs. The ComFoRT interpretation generates a system description in which system behavior is divided into communicating, concurrent modules (or processes[6]) described in CFA. Most behavior is described in C, while FSP expressions are used to describe the manner in which processes communicate with each other via events.

The ComFoRT model checking engine has been used to verify a number of real systems [Chaki 04b].[7] It was used to discover a bug in Micro-C OS version 2.00, a real-time operating system for embedded software consisting of about 3,000 lines of American National Standards Institute (ANSI) C code. It has also been used to verify an extensive set of claims against the OpenSSL implementation, an open source implementation of the Secure Socket Layer protocol used to exchange information over the Internet.

5. Finite-sequential processes (FSP) is a process algebra used to concisely describe state machines composed primarily of patterns of events (i.e., with little persistent state information) [Magee 01].
6. The word *process* is used here with the process algebra meaning—a unit of concurrency—and not to indicate any particular implementation of concurrency, such as a thread or operating-system process.
7. This work was originally implemented and reported with respect to MAGIC.

For simplicity, we will no longer distinguish between what was originally part of MAGIC and what was introduced for the ComFoRT reasoning framework. Both will be considered as part of the resulting ComFoRT model checking engine.

4.1.1 Abstraction-Based Verification Approach

The core feature of ComFoRT that enables it to verify software is the abstraction-based approach to finite-state model extraction. Additionally, ComFoRT has several other important features that make it particularly useful for verifying software:

- It extracts finite-state models from concurrent, message-passing C programs and refines these models using a CEGAR framework.
- Predicate abstraction, counterexample validation, and model refinement are all performed compositionally (i.e., one concurrent unit at a time) [Chaki 03b].
- It uses numerous optimization algorithms that greatly reduce the sizes of the finite-state models that it produces [Chaki 03a].

Given a program made up of a number of concurrent modules (processes) $C_1 \ldots C_n$, a set of predicates on program variables, and a claim φ, ComFoRT automatically extracts abstract models $M_1 \ldots M_n$, respectively, and checks whether the parallel composition of these models satisfies φ. It then uses the results obtained from this verification to refine the models, if necessary, as described below.

Finite-state model extraction. Model extraction uses predicate abstraction to transform infinite-state (or very large) CFA programs into finite-state models that are amenable to model checking. Given a set of predicates defined over the state variables of a system, predicate abstraction constructs an abstract model that describes the behaviors of the original system in terms of these predicates. For example, if *x*, *s*, and *t* are integer variables of our C program, the expressions P := "$x < 5$" and Q := "$s+t = 3$" are two possible Boolean-valued predicates on these variables—that is, each of these predicates can take only two possible values, *True* or *False*. Therefore, the model obtained from predicate abstraction with predicates *P* and *Q* has a finite number of states, whereas the original system has an unbounded number of states since it operates on integer variables.

Once a finite set of predicates is chosen, the states of the corresponding abstract model are simply valuations of the predicates. Using the previous example, there would be four different states corresponding to the different combinations of truth assignments to *P* and *Q*. Each abstract state *A* symbolically represents the set of all the states of the original C program that agree with *A* on the valuations of the predicates. For example, the abstract state A := <*P=True,Q=False*> corresponds to all the C program states where variable *x* is less than 5, and the sum of *s* and *t* is not 3.

The transition relation of the abstract system is defined *existentially*: we postulate a transition from abstract state *A* to abstract state *B* if there are concrete states *a* and *b*, associated to *A* and *B* respectively, with a transition from *a* to *b*. Figure 4 illustrates these ideas.

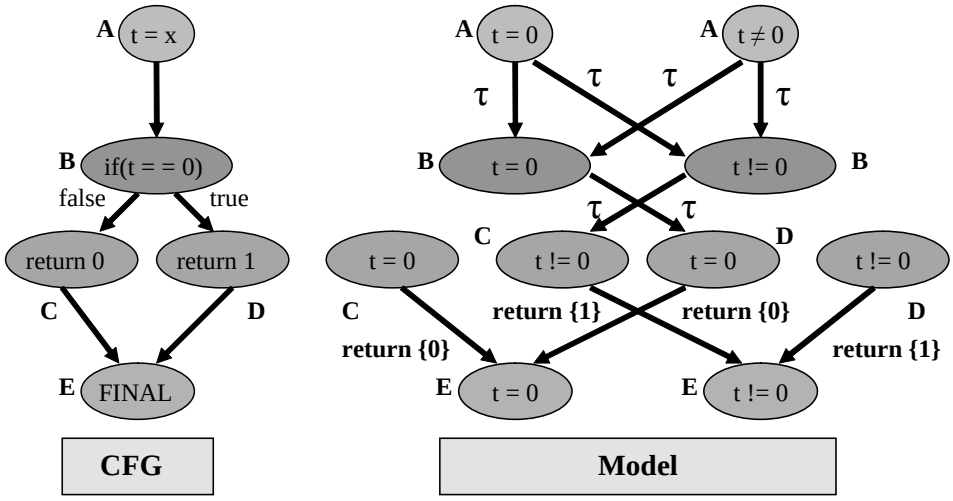

Figure 4: Model Extraction Through Predicate Abstraction

The left-hand side shows the control flow graph (CFG) of a simple C program with two integer variables *x* and *t*. If we define a single predicate *P* := "*t = 0*," two abstract states correspond to each control location: *P* can either be *True* or *False*. The right-hand side of Figure 4 shows the model that we obtain via predicate abstraction. Transitions are labeled with actions that can represent synchronization events (absent here), the return values of procedure calls, and internal actions (τ). Models are composed by synchronizing on shared events and interleaving on other (local or internal) actions. Note that certain abstract states are unreachable (e.g., the state corresponding to location `C` and valuation *True* for *P*). Intuitively, this is true because the program can never take the `else` branch of the `if` statement in location `B` when *t* is equal to zero.

The initial set of predicates can be obtained in many ways. The most common way is to collect formulas appearing in conditional expressions as well as in the claim to be checked. The user can also specify predicates of interest, perhaps based on some deeper understanding of the system. New predicates are generated, if needed, in the model refinement phase, which is described next.

Model refinement using CEGAR. The model constructed by predicate abstraction is guaranteed to be a *conservative* abstraction of the original system, meaning that each behavior in the original system is represented by some behavior in the model, although the model may contain more behaviors. As a result, if the model satisfies the claim, so does the original system [Clarke 94]. However, a counterexample obtained by verifying the model may be spurious. The model checking engine analyzes the counterexample and, if it is

spurious, uses this information to derive additional predicates and construct a new, finer abstraction of the system. The new predicates, which rule out the counterexample in the new model, are obtained automatically using a theorem prover. In choosing the new predicates, algorithms are used that attempt to keep the size of the corresponding abstract model as small as possible. The verification is then repeated with the refined model.

4.1.2 Compositional Verification Approach

In addition to automated abstraction procedures, the model checking engine applies compositional reasoning within the CEGAR framework to further reduce verification complexity. Assume that a program C consists of modules $C_1 \ldots C_n$ executing concurrently. The algorithms that check whether a claim φ holds for C use the following three-step iterative process.

1. **Abstract**. Create an abstract model $M = M_1 \,||\, \ldots \,||\, M_n$. Note that the construction of the M_is can be done one module at a time without constructing the full state space of C. Further, it can be shown that if C has an error, so does M.
2. **Verify**. Check if a claim φ holds for M. If it does, report success and exit. Otherwise, let CE be a counterexample that indicates where φ fails in M.
3. **Refine**. Check whether CE is a valid counterexample with respect to C. Once again, this is done one module at a time. If CE corresponds to a real behavior, the algorithm reports a failure and a fragment of each M_i that shows why $\neg(C \models \varphi)$. If CE is spurious, refine M using CE to obtain a more precise abstract model and repeat from Step 1.

Note that only the verification stage (Step 2) requires the explicit composition of modules, though this composition always involves only the abstract models. All other stages can be performed compositionally (i.e., one module at a time).

This abstract-verify-refine loop continues until a real counterexample is obtained, or the system is verified to be correct. In theory, the CEGAR loop is not guaranteed to terminate when verifying arbitrary CFA programs. In practice, however, it has been quite effective [Clarke 99].

4.1.3 State/Event-Based Formalism

The limited expressiveness of classical temporal logics (discussed in Section 3.2) is particularly restrictive when verifying component-based systems; useful claims often involve patterns of communication among components that are dependent on the state of the participants. To increase the usability of model checking for verification of software designs, new specification and verification techniques were designed and implemented in the ComFoRT model checking engine. One of those techniques is a formalism in which both state-based and event-based claims could be expressed, combined, and verified [Chaki 04b].

The formalism consists of special state transition systems called *labeled Kripke structures* (LKS), which extend the type of finite-state models described in Section 4.1.1. An LKS is a directed graph in which states are labeled with atomic propositions and transitions are labeled with actions (where an event is a particular kind of action). The claim logic that was developed is a state/event derivative of Linear Temporal Logic (LTL). The state/event LTL directly represents both software models and claims without any annotations or privileged insights into system execution. Experiments show that standard, efficient LTL model checking algorithms can be applied, at no extra cost in *space* or *time*, to help reason about state/event-based systems. Earlier noted experiments with OpenSSL and Micro-C OS show that this new approach not only simplifies writing claims, but also displays important gains in space and time during verification. In certain cases, claims were encountered that could not be verified using traditional approaches that were purely event or state based due to state space explosion. These same claims were tractable within the state/event framework.

The state/event-based formalism is suitable for sequential and concurrent systems and preserves the ability to use modular abstraction refinement procedures that are embedded within a CEGAR framework.

4.1.4 Deadlock Detection

Verifying the absence of deadlock in a composed system is a common requirement that must be satisfied before a system can be deployed. This is especially true for safety-critical systems, such as embedded systems or plant controllers, that are always expected to be responsive to external stimuli or service requests within a fixed deadline. Moreover, whenever deadlock is detected, it is highly desirable to be able to provide system designers and developers with feedback showing what caused the deadlock.

However, despite significant efforts, validating the absence of deadlock in systems of realistic complexity remains a major challenge. The problem is especially acute for concurrent processes that communicate via mechanisms with blocking semantics (e.g., synchronous message-passing and semaphores). The primary obstacle is the well-known state space explosion problem. Both abstraction and compositional reasoning, though successful during verification of other claims [Grumberg 94, Clarke 94, Henzinger 00], are less useful in detecting deadlock. This is because deadlock is inherently noncompositional, and its absence is not preserved by standard abstractions.

The ComFoRT reasoning framework uses a model checking approach for deadlock detection [Chaki 04a]. To detect deadlocks, the ComFoRT model checking engine extends the compositional CEGAR framework (described in Section 3.3.3) with a notion of *abstract refusals* to either detect a deadlock or to prove that no deadlock exists. The extensions are grounded in standard process-algebraic theory [Hoare 85]. The resulting CEGAR approach for deadlock detection is completely automated and provides a counterexample whenever a deadlock is detected.

4.2 CCL Designs

Developers describe their components and assemblies using CCL [Wallnau 03b], which captures architectural information in terms of component and interaction mechanism types using behavioral descriptions, topological descriptions of assemblies, and property annotations. How behavior is described in CCL is particularly important to model checking.

In CCL, behavior is described by statecharts based on UML statecharts [OMG 02]. Figure 5 shows a graphical depiction of a CCL statechart for a simple component.

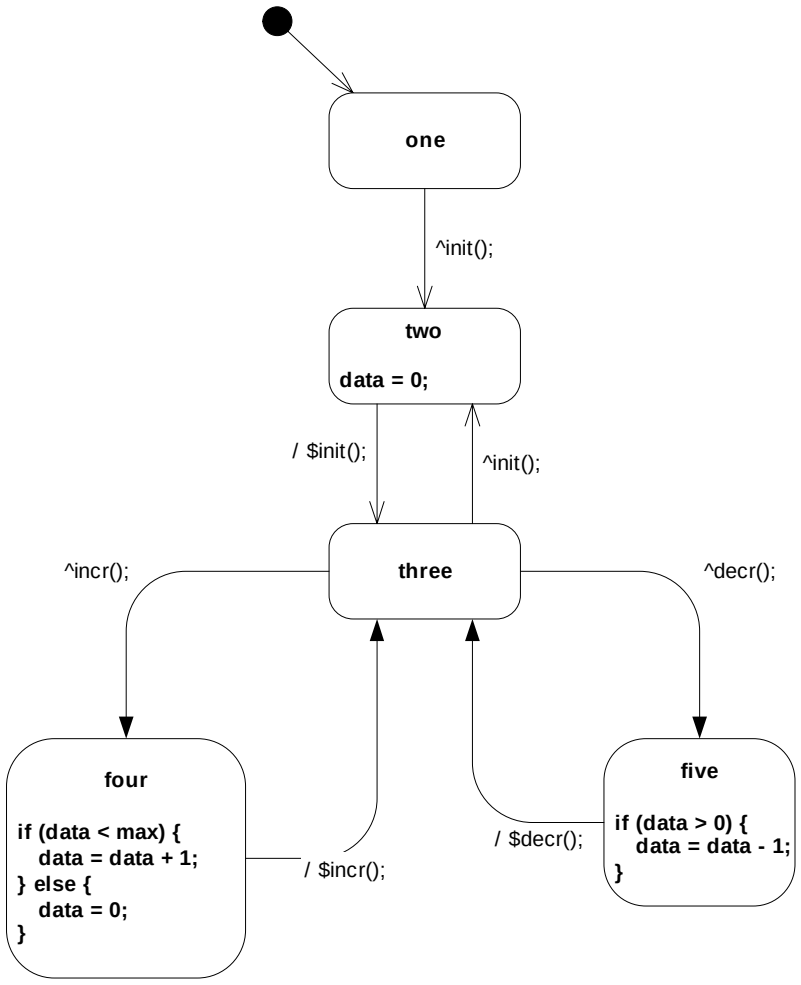

Figure 5: Graphical Depiction of a CCL Statechart

CCL statecharts enrich UML statecharts with modeling conventions that describe how components interact (e.g., the use of ^ and $ in event names to indicate the initiation and completion, respectively, of interactions) and restrict the use of some UML statechart constructs (e.g., hierarchy and history states). Additionally, CCL supplies a C-based action language that defines state and transition actions, much like the action language in xUML [Mellor 02]. CCL fills in the gaps of UML statechart semantics by assigning specific meaning to semantic variation points left open in the UML standard (e.g., whether a statechart's

implicit event queue is FIFO or priority based), resulting in precise semantics that enable sound translation during interpretation.

Retaining UML statechart semantics enables developers who are already familiar with UML to quickly learn and apply CCL.[8] Using a single design language (CCL), rather than a different language for each reasoning framework, also has benefits.

- The same CCL specification is used as the source of all PECT reasoning frameworks, so developers do not have to learn multiple languages and write and maintain multiple specifications.
- Different model checking reasoning frameworks could be applied to the same CCL specification without requiring any developer intervention or changes (e.g., if different model checkers were more effective for different types of claims).

While CCL statecharts are the primary inputs for the ComFoRT interpretation to CFA programs, the interpretation also translates the claims to be checked by the model checking engine. These claims are captured in CCL as property annotations that can be attached to components or assemblies.

CCL specifications can describe an assembly in various levels of detail—from structural (topological) information only, to coarse designs in which only basic control and communication decisions have been made, and finally to rich descriptions in which enough information is present to generate complete component implementations. Developers can use ComFoRT to understand the consequences of however many decisions have been captured in a CCL specification, but ComFoRT is most useful when applied to specifications containing large amounts of concurrency and coordination among components—the consequences of which are difficult to understand for all possible executions.

Applying ComFoRT to design specifications, rather than complete implementations, is preferred. Economically, applying ComFoRT to designs allows errors or inconsistencies to be identified and eliminated prior to implementation or integration, with the inherent cost savings that earlier error detection yields. Technically, applying ComFoRT to designs—which typically abstracts away many details like routine error checking, logging, and data marshalling/unmarshalling—greatly reduces model checking complexity. While such details are needed in robust implementations, their contribution to state space explosion is not commensurate with the benefit, as these details often are not relevant to the architectural issues being analyzed.

8. Using UML statecharts was a natural choice given their expressiveness and widespread use. But we are obligated to avoid semantic surprises—users should never wonder whether a particular symbol has the same meaning in CCL as it has in UML. CCL meets this obligation by not redefining the semantics of statechart constructs.

While one of the objectives of PECT is to impose constraints that guarantee the analyzability of all constructible designs (i.e., all CCL specifications that satisfy the constraints of the component technology and the reasoning framework), we can only guarantee that analysis models can be generated by interpretation from all constructible designs. We cannot guarantee the tractability of the analysis, particularly in the case of model checking and the state space explosion problem. We can, however, impose additional constraints that improve the likelihood that analysis of constructible designs will be tractable. For example, we could impose the constraint that designs may not share variables among threads because shared variables accelerate state space growth.[9]

4.3 Automated Interpretation

Automated interpretation generates CFA programs that correspond to CCL specifications of components and their assemblies. As CCL and CFA are used to describe software at different levels of abstraction, this is a nontrivial task that involves reconciling many similar, but slightly divergent, concepts between CCL and CFA.

In CCL, the architectural units of description are components, interfaces, assemblies, and interaction mechanisms. A component is an independently deployable unit of implementation that may consist of a number of reactions, each of which models some behavior of a system that always executes in the same thread of control (e.g., an actual thread or the body of a function call). Each reaction, as well as each type of interaction mechanism, has an associated statechart description.

In ComFoRT, the units of description are much lower level and much simpler—a program consists of a number of concurrent processes, each of which has its own state machine description expressed in terms of C and FSP. Processes communicate by synchronizing on occurrences of shared events. CCL concepts like components and interaction mechanism types have no explicit representation in CFA programs, but are instead represented explicitly in terms of more primitive modeling elements.

Bridging the gap between a CCL specification and a CFA program involves resolving a lot of little issues. Many of the issues are easy to resolve and can be resolved many ways. However, many of the decisions impact the choices that are available for resolving other issues, and resolving the issues *consistently* turns out to be a harder problem. The remainder of this section summarizes a variety of the issues handled by the ComFoRT interpretation. It includes three categories of significant issues and miscellaneous simpler issues. The significant issues are

9. This constraint is not currently imposed because it is too limiting for the types of systems we currently work with—reactive, real-time controllers. However, it is an excellent example of the type of constraints that could be imposed to improve tractability.

1. supplying ComFoRT-relevant information that PECT users are not required to provide
2. determining the right set of processes to model in CFA to represent the concurrency found in a collection of reaction and interaction mechanism specifications in CCL
3. determining how to translate from CCL statecharts to CFA programs

While all of these decisions could be solved without a PECT, formalizing them in an automated interpretation has several benefits. All decisions are made once by the experts developing the reasoning framework for the PECT who best understand the subtleties involved. Users do not have to ponder the issues and can focus on what they care about—their components and assemblies. Automated interpretation prevents human errors in application and can be proven once, rather than for each CFA program produced.

4.3.1 Supplying Relevant Information

Some information that is relevant to the execution of the software assembly does not depend on the system components. The interaction mechanisms and services provided by the component technology's runtime environment are good examples. Interaction mechanism semantics—for example, queuing and blocking policies—certainly affect how an assembly behaves, though the mechanisms are not developed by PECT users and, hence, are not something users should be expected to know well enough to model formally.

However, the interpretation requires specifications for all behaviors contributing to assembly behavior, which includes interaction mechanisms and services. Rather than passing this requirement on to users, the construction framework of the PECT includes a library of specifications for the types of interaction mechanisms and services provided by its component technology. These specifications are incorporated and specialized automatically as part of the ComFoRT interpretation. The specializations range from the simple (e.g., name matching) to the complex (e.g., modifying interaction mechanism statecharts at well-defined variation points to account for a component's use of multiple, related interaction mechanisms).

4.3.2 Determining Processes

Deciding how to map reactions and interactions to processes in a CFA program is relatively straightforward because there is a convenient benchmark for comparison—the executing software. To correctly model implementation concurrency, each process in the generated CFA program should correspond, one to one, with a thread of execution in the implementation. Correct modeling of implementation concurrency is crucial since incorrect modeling can cause model checking to produce incorrect results.

Overapproximating a model's implementation concurrency (i.e., allocating behavior that cannot execute concurrently to multiple processes) leads to spurious counterexamples that cannot occur in the executing system. Deciding whether any given counterexample is

spurious can be a time-consuming task and is particularly wasteful when a counterexample is found to be spurious. Underapproximating a model's implementation concurrency (i.e., allocating behavior that can execute concurrently to the same process) is more insidious, since it prevents the discovery of counterexamples that can occur in the executing system.

To eliminate one source of overapproximation and underapproximation, the ComFoRT interpretation determines the implementation concurrency from information in the CCL specifications and knowledge of the PECT's component technology. The interpretation then composes reaction and interaction statecharts into a set of processes corresponding to the implementation concurrency. Roughly, this composition involves the following steps:

1. Sequentially compose a copy of each nonthreaded reaction with every reaction that initiates an interaction with it (recursively through the assembly).
2. Remove all original nonthreaded reactions.
3. Compose the remaining reactions, which correspond to threads of execution in the implementation, in parallel.
4. Compose the interactions (for those interaction types requiring separate statecharts) in parallel with the remaining reactions.

The result is a CFA program that includes one process for each thread of control along with one process for each interaction (for specific types of interactions). An example is shown in Figure 6 and Figure 7.

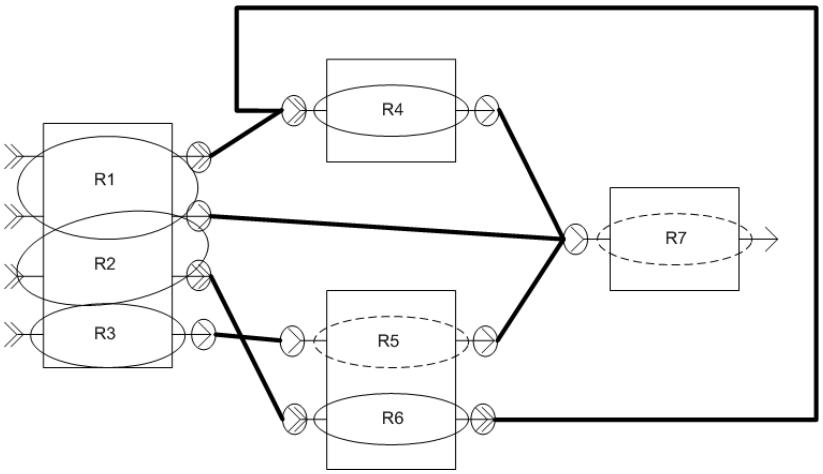

Figure 6: Graphical Depiction of a CCL Assembly and Its Reactions

Figure 6 shows a graphical depiction of a CCL specification in which boxes are components, arrow ends are interfaces, thick lines are interactions between components via the connected interfaces, and ovals are reactions (solid lines for threaded reactions, dashed lines for nonthreaded).

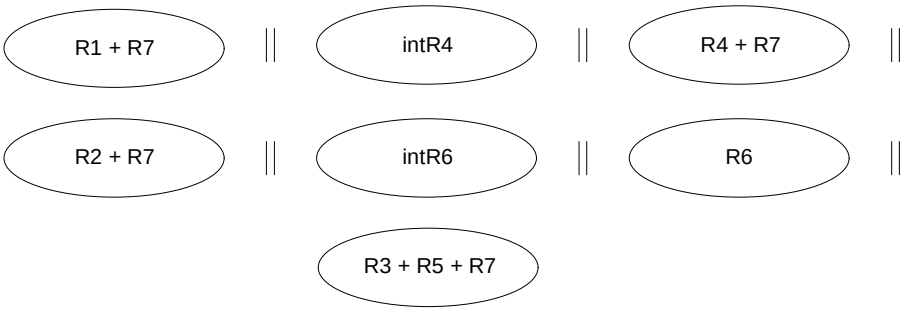

Figure 7: Graphical Depiction of Processes in the Generated CFA Program

Figure 7 shows the processes in the corresponding generated CFA program. Each process labeled with some combination of reaction names (e.g., R1 + R7) represents a thread of control in the executing system and the set of reactions that execute within that thread of control. The remaining processes (e.g., intR4) are those introduced to capture the behavior of interaction types requiring statecharts (e.g., intR4 models the message queue for messages sent to R4).

This composition process almost achieves our goal of correctly modeling the implementation concurrency; the manner in which interaction behavior is currently incorporated (as additional processes) violates our goal of having each process in a CFA program correspond, one to one, with a thread of execution in the implementation.

While behavioral specifications are needed for the interaction mechanisms and modeling each as an independent process in a CFA program is convenient, interaction mechanisms are not realized by separate threads of control in many runtime environments. Therefore, they should not always be modeled as separate processes. The consequence of the current solution is an overapproximation of concurrency, which does not prevent the discovery of counterexamples but may introduce spurious ones. Alternative solutions, as well as a better understanding of the consequences of the current solution, are an open area of investigation.

For more details on this approach to systematically compose reactions and correctly model implementation concurrency, see the work of Ivers and Wallnau [Ivers 03].

4.3.3 CCL Statechart – CFA Program Translation

Of all the issues involved in automated interpretation, translating a CCL statechart to a CFA program, as specified in C and FSP, is the most difficult task. Both languages represent state transition systems, but they have different syntactic elements and sometimes assign different semantics to syntactic elements with the same name. While a full explanation of all the differences and the details of the translation are beyond the scope of this report, the following sections describe some of the more important translation issues.

Communication semantics. The first (and perhaps biggest) difference is the semantics of communication between state machines. While both computational models are state machines that communicate by exchanging events, their mechanisms are quite different.

- Statechart semantics allow the sending and receiving of an event to be independent activities. That is, when a state machine sends an event to another state machine, that event is queued for later delivery to the second state machine, and the first state machine is allowed to proceed independently of the eventual delivery of the event.
- The communication semantics of CFA, based largely on FSP semantics,[10] require the sending and receiving of an event to be simultaneous activities. That is, a sender can only send an event when the receiver is prepared to accept that event, and a receiver can only receive an event when it is sent by a sender—that is, event exchange is a synchronized activity in which two processes *engage* in an event at the same time.

Consequently, CFA programs must be generated that capture statechart semantics. Statecharts have an implicit event queue for each state machine that effectively decouples sender and receiver progress and defines semantics for a receiver with multiple queued inputs (e.g., in terms of order of delivery).

An implicit event queue is represented explicitly in the generated CFA programs by a process that is always ready to accept an event from a sender without immediately involving the receiver. This allows senders in CFA programs to synchronize with an intermediary without delay (i.e., without receiver involvement), modeling the looser restrictions on senders in statecharts. Interaction mechanism specifications, which describe the mediation of inter-component communication, model this behavior in most cases.[11] Further, generated reaction and interaction models use a particular event synchronization protocol to complete statechart communication semantics—ensuring that reactions receive events only at appropriate states.

Shared data. A second, related difference is that statecharts provide forms of communication that are not currently available in the model checking engine. In statecharts, two concurrent state machines can communicate via shared data. Since the model checking engine does not support shared data, such communication must be modeled by means that it does support—event communication. The current means to model this using CFA (though not the most efficient approach) is to model shared data in a separate process always ready to accept events representing read or write operations. Data accesses from statecharts are translated as read and write events in CFA programs that communicate with the shared data process. Such conceptual transformations are performed by the interpretation as needed.

10. Communication semantics for FSP are, in turn, based on the communication semantics of CSP [Hoare 85].
11. There are some cases where topological knowledge eliminates the need for such an explicit queue. Unthreaded reaction models, due to our composition algorithm, never accept events from more than one reaction and always block the sender until processing is complete. As a result, this can be modeled accurately using the communication semantics of CFA without an explicit queue.

Unconstrained events. A third issue involves the different way in which statecharts and CFA treat unconstrained events. In a CFA program, a process that engages in an event that is not shared with any other process with which it is composed is unconstrained with respect to that event and may engage in the event at any time. In statecharts, a state machine that consumes an event that is not generated by any other state machine with which it is composed will never receive that event. Consequently, the interpretation looks for unconstrained events in the statecharts, particularly interface events representing component behaviors not used in a particular assembly, and models the statechart semantics by excluding the events (and the behavior they trigger) from the generated CFA programs.

Event names. A fourth issue involves the uniqueness of event names. In CCL, multiple reactions of the same component can initiate interactions on the same interface, representing the ability to use the same resource independently.[12] To model this, the statechart for each reaction independently sends an event with the same name to the same receiver. In a CFA program, however, when multiple processes can engage in the same event, each time any one of them engages in the event, they must all engage in the event.

This imposes a very different meaning—for example, if one reaction tries to call a function that another reaction can call, they must call it together, using the same parameters and getting the same result. To retain the statechart meaning, events are renamed during interpretation such that no two processes can send the same event and interactions are written to accept several different events, each of which has the same logical meaning.

Generating equivalent control structures. The last issue is deciding how to generate a CFA program with a control structure that is equivalent to a CCL statechart. Since C is the central portion of the CFA input language, we take hints from existing techniques used to generate code from state machines. Each process in the CFA program is structured as follows:

- A C function is generated for each process (as shown by the `store_S` function in Figure 8).
- Each function has an integer variable recording the current state, where the values correspond to the states in the CCL statechart (as shown by the `currState` variable and the `#define` statements in Figure 8).
- Each function is organized around a `while` loop containing a `switch` statement in which each `case` corresponds to one of the states in the CCL statechart (as shown in Figure 8).
- State changes (transitions) are represented as assignments to the state variable, followed by leaving the `switch`, reentering the loop, and then reevaluating the `switch` to enter the `case` for the new state (as shown in Figure 9).

12. At least, independently from the perspective of the sending reactions' component. Interactions with a common receiving reaction are still subject to any coordination provided by the interaction mechanism, such as mutually exclusive access.

```
component store() {
   ...
   threaded react S(init,incr,decr) {
      state two {  // init
      ...
      }

      state four { // incr
      ...
      }

      state five { // decr
      ...
      }

      start -> one {}
      one -> two {trigger ^init();}
      two -> three {action $init();}
      three -> two {trigger ^init();}
      three -> four {trigger ^incr();}
      four -> three {action $incr();}
      three -> five {trigger ^decr();}
      five -> three {action $decr();}
   } // end of react S
} // end of component store
```

```
#define  START  0
#define _one_ 1
#define _two_ 2
#define _three_ 3
#define _four_ 4
#define _five_ 5

void store_S() {
   int currState;
   ...
   currState = 1;

   while(1) {
      switch(currState) {
      case _one_:
         ...
      case _two_:
         ...
      case _three_:
         ...
      case _four_:
         ...
      case _five_:
         ...
      } // switch (currState)
   } // while (1)
}
```

Figure 8: Fragment of a CCL Specification (Left) and a CFA Program (Right) Illustrating Equivalent Control Structures

```
component store() {
   ...
   threaded react S(init,incr,decr) {
      ...
      three -> two {trigger ^init();}
      three -> four {trigger ^incr();}
      three -> five {trigger ^decr();}
      ...
   } // end of react S
   ...
} // end of component store
```

```
void store S() {
   ...
   while(1) {
      switch(currState) {
      ...
      case _three_:
         ...
         nextEvent =
fsp_S_externalChoice();
         switch (nextEvent) {
         case __decr__:
            currState = _five_;
            break; // case __decr__
         case __incr__:
            currState = _four_;
            break; // case __incr__
         case __init__:
            currState = _two_;
            break; // case __init__
         } // switch (nextEvent)
         break; // case _three_
      ...
      } // switch (currState)
   } // while (1)
}
```

Figure 9: Fragment of a CCL Specification (Left) and a CFA Program (Right) Illustrating Equivalent Control Structures with an Emphasis on Transitions

While the translation of action statements (like comparisons and arithmetic operations) is straightforward (as shown in Figure 10), the treatment of communication events is not. The C language does not include events as syntactic elements; therefore, the model checking engine delegates event handling to FSP expressions related to the C code by C functions with the same names. Calling one of these functions in the C code instructs the model checking engine to use the corresponding FSP expression at that location when combining the models to generate an underlying state transition system representation (more precisely, an LKS).

```
component store() {
   ...
   threaded react S(init,incr,decr) {
      ...
      state four { // incr
         entry {
            if (data < max) {
               data = data + 1;
            } else {
               data = 0;
            }
         }
      }
      ...
   } // end of react S
   ...
} // end of component store
```

```
void store S() {
   ...
   while(1) {
      switch(currState) {
      ...
      case _four_:
         if (store_data < store_max) {
            store_data = store_data +
1;
         } else {
            store_data =  0;
         }
         ...
         break; // case _four_
      ...
      } // switch (currState)
   } // while (1)
}
```

Figure 10: Fragment of a CCL Specification (Left) and a CFA Program (Right) Illustrating Equivalent Control Structures with an Emphasis on Actions

A statechart uses events in two ways: as transition triggers (i.e., communication that is received) and as actions (i.e., communication that is sent). A sent event is translated as a function call bound to an FSP expression that sends the appropriate event. A received event, however, cannot always be translated in isolation. Often, a statechart can transition from a state by one of multiple transitions, each of which may have a different event as a trigger. This behavior is translated as a function call bound to an FSP expression that allows a choice of events and returns information to the C program indicating which event occurred (see the CFA text in Figure 11 that corresponds to the CCL specification fragment from Figure 9).

```
// ---------- Sink Pin (Event) Defines
#define __decr__ 0
#define __incr__ 1
#define __init__ 2

void store_S() {
   ...
   while(1) {
      switch(currState) {
      ...
      case _three_:
         ...
         nextEvent =
fsp_S_externalChoice();
         switch (nextEvent) {
         case __decr__:
            currState = _five_;
            break; // case __decr__
         case __incr__:
            currState = _four_;
            break; // case __incr__
         case __init__:
            currState = _two_;
            break; // case __init__
         } // switch (nextEvent)
         break; // case _three_
      ...
      } // switch (currState)
   } // while (1)
}
```

```
cproc fsp S externalChoice {
   abstract {abs, 1, S_ExtChoice};
}
S_ExtChoice=(
   begin_decr -> return {$0 == 0} ->
STOP |
   begin_incr -> return {$0 == 1} ->
STOP |
   begin_init -> return {$0 == 2} ->
STOP).
```

Figure 11: Fragment of a CFA Program Illustrating How C (Left) and FSP (Right) Are Combined (Through fsp_S_externalChoice) to Represent Receiving Events

While this approach to modeling a statechart in C and FSP is not the most efficient, it is easily verified and allows us to make progress on the broader range of issues involved in interpretation while we consider more efficient alternatives. A complete example is found in Appendix A.

4.3.4 Miscellaneous Issues

Miscellaneous and comparatively mundane issues are also handled by automated interpretation in ComFoRT.

Scoping rules. CCL and CFA have different scoping rules. In CCL, components and reactions have their own related namespaces, allowing a reaction and its component to declare variables with the same name. Both variables are visible within the inner scope (the reaction), and syntactic means are provided to unambiguously refer to one or the other. CFA, however, has a single, flat namespace for each process. Therefore, the interpretation performs simple name mangling to avoid name collisions in CFA programs.

Translating claims. Claims to be checked also appear in CCL and must be translated for use with the model checking engine. Claims are written in a temporal logic and refer to event and variable names as they appear in the CCL specifications. However, events and variables are renamed during interpretation, and the same renaming scheme is applied to claims. While largely straightforward, handling events requires more than just renaming. When event renaming introduces multiple distinct names in CFA programs for the same statechart event, claims must be rewritten to allow a choice among the renamed events.

Determining seed predicates. The interpretation determines the set of seed predicates that should be included in the CFA program to guide the model checking engine's automated predicate abstraction algorithms and reduce abstraction costs. Currently, a useful but not terribly sophisticated strategy is implemented. The predicates used are extracted from the control points in the generated CFA program that represent the structure of the state machine (e.g., the possible values of the current state variable and other variables controlling transition among states).

Selecting options. The interpretation determines which set of options should be used when invoking the model checking engine in ComFoRT to check each claim. A claim may be checked more or less efficiently based on the characteristics of the model and the claim and the choice of algorithms applied during model checking. The model checking engine allows some algorithm choices and parameters affecting the algorithms to be given as command line options. For each claim to be checked, the interpretation supplies a suitable set of options to be used during model checking.

4.4 Reverse Interpretation

The claims to be checked are recorded in CCL as the names of property annotations. Until the claim has been checked, however, these annotations have no value. The automated reverse interpretation takes the model checking results and stores the corresponding CCL representation as the value of the annotation. Model checking results for a claim always take one of following three forms:

1. The claim is true.
2. The claim is false and a counterexample under which it is false is provided.
3. The claim cannot be evaluated due to state space explosion.

The first and third options are easy to translate for use as CCL property annotations. The second option is the most interesting one.

A counterexample is a sequence of execution steps needed to reach a state in which a claim is false. In ComFoRT, this is a list of statements from the CFA program written in C and FSP, each of which represents an event or action in one (or more in the case of event communication) of the CFA program's processes. Reading a counterexample is much like stepping through a debugger—you observe a particular sequence of variable manipulations, condition evaluations, and function calls (sent or received events via the corresponding FSP expressions).

However, each statement is from the CFA program and not the CCL specifications, so all differences discussed for interpretation apply:

- The processes do not correspond one to one with CCL reactions.
- The CFA program includes information not found in CCL specifications (e.g., event queues that are implicit in statecharts).
- Some CCL concepts have been modeled differently (e.g., shared data as event communication).
- States are represented by the value of a variable in a CFA program, not the text label from CCL.
- CFA program variable and event names are different from those in the CCL specifications.

Given these differences, users could not be expected to understand a CFA-based counterexample without also understanding the semantics of CFA and the transformations performed by the interpretation. To package such complexity, ComFoRT includes a reverse interpretation: CFA-based counterexamples are converted to a sequence of statements expressed in terms of CCL specifications by reversing the types of transformations used to generate a CFA program.

5 Current Status and Next Steps

A PECT packages the complexities of a quality-specific analysis technique in a reasoning framework that can then be used to predict the behavior of assemblies of components. The ComFoRT reasoning framework packages the effectiveness of a state-of-the-art model checking engine in a form that enables users to apply the formal analysis technique without being experts in its use. Initial applications in verifying the safety claims of a small industrial example were effective and uncovered a significant concurrency problem in which processes could be unnecessarily blocked.

Future steps include extending the ComFoRT reasoning framework in various directions to

- Expand the component technology features that can be described in CCL.
- Scale the application of ComFoRT to larger assemblies of components.
- Address component technology specific verification problems in addition to verification of individual user-supplied claims.
- Further reduce the expertise required to apply ComFoRT.

Component technology features are already being expanded to include, among other things, fully automated interpretation and reverse interpretation. We are also working to enhance data communication support, including parameterized events and shared data.

Scaling the size of assemblies that can be checked by ComFoRT involves two kinds of activities. First, we will continue to research compositional reasoning techniques, targeted abstractions, and other procedures that alleviate state space explosion. Second, we can optimize the CFA programs generated by interpretation such that they are more efficiently verified (e.g., by generating programs with fewer control points, resulting in fewer predicates).

To address component technology specific problems, we plan to extend the compositional deadlock detection work and develop algorithms for checking (certifying) whether a component's CCL statechart specification is consistent with its implementation.

Finally, we are considering two ways to further reduce the expertise required to apply ComFoRT. First, we could provide developers with a pattern-based approach to specifying claims (like the approach described by Dwyer, Avrunin, and Corbett [Dwyer 99]), rather than requiring developers to use a temporal logic to write claims. Second, we could improve the usefulness of predictions made by ComFoRT by eliminating spurious counterexamples stemming from remaining sources of concurrency overapproximation (e.g., the lack of an

explicit scheduler in generated CFA programs) and by presenting counterexamples in a more user-friendly form.

Appendix A Interpretation Example

This section contains the complete input and output of the ComFoRT interpretation for a simple example. This example contains a single component with a single reaction whose behavior was shown graphically in Figure 5. As it is a simple example, it does not illustrate all interpretation concepts explained in Section 4.3, but many of the concepts from Sections 4.3.3 and 4.3.4 are shown.

Section A.1 contains the CCL specification used as input to the ComFoRT interpretation. For more information on CCL syntax, see Wallnau and Ivers' work [Wallnau 03b].

Section A.2 contains the two files that make up the generated CFA program. For more information on CFA syntax (a feature of MAGIC retained in the ComFoRT model checking engine), see the MAGIC tutorial at http://www.cs.cmu.edu/~chaki/magic/tutorial-1.0.html.

A.1 Input: CCL Specification (store.ccl)

```
component store() {
   int data = 0;
   int max = 5;

   sink asynch init();
   sink asynch incr();
   sink asynch decr();

   threaded react S(init,incr,decr) {
      state two {    // init
         entry data = 0;
      }

      state four {   // incr
         entry {
            if (data < max) {
               data = data + 1;
            } else {
               data = 0;
            }
         }
      }

      state five {   // decr
         entry {
            if (data > 0) {
               data = data - 1;
            }
         }
      }

      start -> one {}
```

```
      one -> two {trigger ^init();}
      two -> three {action $init();}
      three -> two {trigger ^init();}
      three -> four {trigger ^incr();}
      four -> three {action $incr();}
      three -> five {trigger ^decr();}
      five -> three {action $decr();}
   } // end of react S

   // claims to be checked
   annotate S {"comfort", const string Claim1 = "G([data >= 0])"}
   annotate S {"comfort", const string Claim2 = "G([data <= max])"}
   annotate S {"comfort", const string Claim3 = "G([data <= 3])"}
   annotate S {"comfort", const string Claim4 = "G([data == max] &
^incr => F([data == 0]))"} // true
   annotate S {"comfort", const string Claim5 = "G([data == max] &
^decr => F([data == 0]))"} // false
   annotate S {"comfort", const string Claim6 = "G([data < 0])"}
} // end of component store
```

A.2 Output: A CFA Program

The generated CFA program consists of store.c and store.spec: store.c contains the C code portion of the program, and store.spec contains the FSP expressions and auxiliary statements (e.g., the claims to be checked). The C code found in store.c is not the final version given to the model checking engine for verification; store.c must be preprocessed to normalize the control structure and perform macro substitution before its use with the model checking engine. The unpreprocessed version is presented here because it is much easier to read and trace to the corresponding CCL specification.

store.c

```
// -------------------- Reaction Defines S:store--------------------

// -------------------- State Defines
#define _START_ 0
#define _one_ 1
#define _two_ 2
#define _three_ 3
#define _four_ 4
#define _five_ 5

// -------------------- Sink Pin (Event) Defines
#define __decr__ 0
#define __incr__ 1
#define __init__ 2

// -------------------- Reaction store:S-----------------------

void store_S() {


// -------------------- declarations ----------------------
   int currState;
   int nextEvent;
   int listening;

//--------------------declare component or service state variables--
   int store_data;
```

```
   int store_max;

//--------------------declare reaction state variables--------------

// -------------------- initialization -----------------------
   store_data = 0;
   store_max = 5;
   // no exit action on START
   // no transition action on START -> one
   currState = 1;

// -------------------- state machine -----------------------
   while(1) {
      switch(currState) {
      case _one_:
         // no entry action for one
         listening = 1;
         while(listening) {
            nextEvent = fsp_S_externalChoice();
            switch (nextEvent) {
            case __decr__:
               // Discard interaction on ^decr()
               fsp_end_decr(); // an action specified in FSP
               break; // case __decr__
            case __incr__:
               // Discard interaction on ^incr()
               fsp_end_incr(); // an action specified in FSP
               break; // case __incr__
            case __init__:
               // Consume interaction on ^init()
               listening = 0;
               // no exit action on one
               // no transition action on one->two
               currState = _two_;
               break; // case __init__
            } // switch (nextEvent)
         } // while (listening)
         break; // case _one_

      case _two_:
         store_data =  0 ;
         // no exit action on two
         fsp_end_init();
         currState = _three_;
         break; // case _two_

      case _three_:
         // no entry action for three
         listening = 1;
         while(listening) {
            nextEvent = fsp_S_externalChoice();
            switch (nextEvent) {
            case __decr__:
               // Consume interaction on ^decr()
               listening = 0;
               // no exit action on three
               // no transition action on three->five
               currState = _five_;
               break; // case __decr__
            case __incr__:
               // Consume interaction on ^incr()
               listening = 0;
               // no exit action on three
               // no transition action on three->four
               currState = _four_;
```

```
               break; // case __incr__
            case __init__:
               // Consume interaction on ^init()
               listening = 0;
               // no exit action on three
               // no transition action on three->two
               currState = _two_;
               break; // case __init__
            } // switch (nextEvent)
         } // while (listening)
         break; // case _three_

      case _four_:
         {
            if ( store_data < store_max )
               {
                  store_data = store_data +  1 ;
               }
            else
               {
                  store_data =  0 ;
               }
         }

         // no exit action on four
         fsp_end_incr();
         currState = _three_;
         break; // case _four_

      case _five_:
         {
            if ( store_data > 0 )
               {
                  store_data = store_data -  1 ;
               }
         }

         // no exit action on five
         fsp_end_decr();
         currState = _three_;
         break; // case _five_

      } // switch (currState)
   } // while (1)
}
```

store.spec

```
cprog my_prog = store_S {
   abstract abs0, { P0::store_data == 0 && P0::store_max == 5 },
Claim10;
   abstract abs1, { P0::store_data == 0 && P0::store_max == 5 },
Claim21;
   abstract abs2, { P0::store_data == 0 && P0::store_max == 5 },
Claim32;
   abstract abs3, { P0::store_data == 0 && P0::store_max == 5 },
Claim43;
   abstract abs4, { P0::store_data == 0 && P0::store_max == 5 },
Claim54;
   abstract abs5, { P0::store_data == 0 && P0::store_max == 5 },
Claim65;
}

// Claim1: G([data >= 0])
Claim10 :  G ([P0::store_data >= 0]);
```

```
// Claim2: G([data <= max])
Claim21 :  G ([P0::store_data <= P0::store_max]);

// Claim3: G([data <= 3])
Claim32 :  G ([P0::store_data <= 3]);

// Claim4: G([data == max] & ^incr => F([data == 0]))
Claim43 :  G ([P0::store_data == P0::store_max] & begin_incr =>  F
([P0::store_data == 0]));

// Claim5: G([data == max] & ^decr => F([data == 0]))
Claim54 :  G ([P0::store_data == P0::store_max] & begin_decr =>  F
([P0::store_data == 0]));

// Claim6: G([data < 0])
Claim65 :  G ([P0::store_data < 0]);


cproc store_S {
   predicate (currState == 1), (currState == 2), (currState == 3),
      (currState == 4), (currState == 5);
   predicate (nextEvent == 0), (nextEvent == 1), (nextEvent == 2);
   predicate (listening);
}


cproc fsp_S_externalChoice {
   abstract {abs, 1, S_ExtChoice};
}
S_ExtChoice=(
   begin_decr -> return {$0 == 0} -> STOP |
   begin_incr -> return {$0 == 1} -> STOP |
   begin_init -> return {$0 == 2} -> STOP).


cproc fsp_begin_decr {
   abstract{ abs, 1, Begindecr };
}
Begindecr=(begin_decr -> return{} -> STOP).


cproc fsp_end_decr {
   abstract{ abs, 1, Enddecr };
}
Enddecr=(end_decr -> return{} -> STOP).


cproc fsp_begin_incr {
   abstract{ abs, 1, Beginincr };
}
Beginincr=(begin_incr -> return{} -> STOP).


cproc fsp_end_incr {
   abstract{ abs, 1, Endincr };
}
Endincr=(end_incr -> return{} -> STOP).


cproc fsp_begin_init {
   abstract{ abs, 1, Begininit };
}
Begininit=(begin_init -> return{} -> STOP).
```

```
cproc fsp_end_init {
   abstract{ abs, 1, Endinit };
}
Endinit=(end_init -> return{} -> STOP).
```

References

URLs are valid as of the publication date of this document.

[0-In 03] 0-In Design Automation. *0-In Formal Verification Products Validate Complex National Semiconductor Bus Bridge Design.* http://www.0-in.com/news/PR_030106_0in.html (2003).

[Abadi 95] Abadi, M. & Lamport, L. "Conjoining Specifications." *ACM Transactions on Programming Languages and Systems 17,* 3 (May 1995): 507-534.

[Abadir 03] Abadir, M.; Albin, K.; Havlicek, J.; Krishnamurthy, N.; & Martin, A. "Formal Verification Successes at Motorola." *Formal Methods in System Design 22,* 2 (March 2003): 117-123.

[Alur 96] Alur, R. & Henzinger, T. A. "Reactive Modules," 207-218. *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. Brunswick, NJ, July 27-30, 1996. Los Alamitos, CA: IEEE Computing Society Press, 1996.

[Bachmann 00] Bachmann, F.; Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; & Wallnau, K. *Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition* (CMU/SEI-2000-TR-008, ADA379930). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html

[Ball 01a] Ball, T.; Majumdar, R.; Millstein, T.; & Rajamani, S. K. "Automatic Predicate Abstraction of C Programs," 203-213. *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI).* Snowbird, Utah, June 20-22, 2001. New York, NY: Association for Computing Machinery (ACM), 2001.

[Ball 01b] Ball, T. & Rajamani, S. K. “Automatically Validating Temporal Safety Properties of Interfaces,” 103-122. *Model Checking Software: 8th International SPIN Workshop* (in Lecture Notes in Computer Science [LNCS], volume 2057). Toronto, Ontario, Canada, May 19-20, 2001. Berlin, Germany: Springer-Verlag, 2001.

[Ball 04] Ball, T.; Cook, B.; Levin, V.; & Rajamani, S. “SLAM and Static Driver Verifier: Technology Transfer of Formal Methods Inside Microsoft,” 1-20. *Integrated Formal Methods: 4th International Conference, IFM 2004.* Canterbury, Kent, UK, April 4-7, 2004. New York, NY: Springer-Verlag, 2004.

[Barakatain 01] Barakatain, L. & Tahar, S. “Model Checking of the Fairisle ATM Switch Fabric Using FormalCheck,” 907-912. *Proceedings of IEEE Canadian Conference on Electrical & Computer Engineering*. Toronto, Ontario, Canada, May 13-16, 2001. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc. (IEEE), 2001.

[Bass 03] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Second Edition*. Boston, MA: Addison-Wesley, 2003.

[Ben-David 03] Ben-David, S.; Eisner, C.; Geist, D.; & Wolfsthal, Y. “Model Checking at IBM.” *Formal Methods in System Design 22*, 2 (March 2003): 101-108.

[Chaki 03a] Chaki, S.; Clarke, E.; Groce, A.; & Strichman, O. “Predicate Abstraction with Minimum Predicates,” 19-34. *Correct Hardware Design and Verification Methods: 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*. L’Aquila, Italy, October 21-24, 2003. New York, NY: Springer, 2003.

[Chaki 03b] Chaki, S.; Ouaknine, J.; Yorav, K.; & Clarke, E. “Automated Compositional Abstraction Refinement for Concurrent C Programs: A Two-Level Approach.” *SoftMC 2003: Workshop on Software Model Checking* (in Electronic Notes in Theoretical Computer Science [ENTCS], volume 89). Boulder, Colorado, July 14, 2003. New York, NY: Elsevier Science, 2003. http://www1.elsevier.com /gej-ng/31/29/23/141/23/26/89.3.004.pdf (2003).

[Chaki 04a] Chaki, S.; Clarke, E.; Ouaknine, J.; & Sharygina, N. "Automated, Compositional and Iterative Deadlock Detection," 201-210. *Proceedings of Formal Methods and Models for Codesign (MEMOCODE '04)*. San Diego, CA, June 23-25, 2004. Madison, WI: Omnipress, 2004.

[Chaki 04b] Chaki, S.; Clarke, E.; Ouaknine, J.; Sharygina, N.; & Sinha, N. "State/Event-Based Software Model Checking," 128-147. *Integrated Formal Methods 4th International Conference (IFM 2004)* (in Lecture Notes in Computer Science [LNCS], volume 2999). Canterbury, Kent, UK, April 4-7, 2004. Berlin, Germany: Springer-Verlag, 2004.

[Chandra 02] Chandra, S.; Godefroid, P.; & Palm, C. "Software Model Checking in Practice: An Industrial Case Study," 431-441. *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*. Orlando, FL, May 19-25, 2002. New York, NY: Association for Computing Machinery (ACM), 2002.

[Clarke 82] Clarke, E. M. & Emerson, E. A. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," 52-71. *Proceedings of the Logics of Programs Workshop* (in Lecture Notes in Computer Science [LNCS], volume 131). Yorktown Heights, NY, May 1981. New York, NY: Springer-Verlag, 1982.

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

[Clarke 89] Clarke, E. M.; Long, D. E.; & McMillan, K. L. "Compositional Model Checking," 353-362. *Proceedings of the 4th Annual Symposium on Logic in Computer Science*. Pacific Grove, CA, June 5-8, 1989. Washington, DC: IEEE Computing Society Press, 1989.

[Clarke 92] Clarke, E. M.; Grumberg, O.; & Long, D. "Model Checking and Abstraction," 343-354. *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Albuquerque, NM, January 19-22, 1992. New York, NY: Association for Computing Machinery (ACM), 1992.

[Clarke 94] Clarke, E. M.; Grumberg, O.; & Long, D. E. "Model Checking and Abstraction." *ACM Transactions on Programming Languages and Systems 16*, 5 (September 1994): 1512-1542.

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

[Clarke 00] Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; & Veith, H. "Counterexample-Guided Abstraction Refinement," 154-169. *Proceedings of the Computer Aided Verification 12th International Conference (CAV 2000)* (in Lecture Notes in Computer Science [LNCS], volume 1855). Chicago, IL, July 15-19, 2000. Berlin, Germany: Springer-Verlag, 2000.

[Dams 94] Dams, D.; Grumberg, O.; & Gerth, R. "Abstraction Interpretation of Reactive Systems: Abstractions Preserving ACTL*, ECTL*, and CTL*," 573-592. *Proceedings of the IFIP TC2/WG2.1/WG2.2 /WG2.3 Working Conference on Programming Concepts, Methods, and Calculi (PROCOMET '94)*. San Miniato, Italy, June 6-10, 1994. New York, NY: Elsevier, 1994.

[Dwyer 99] Dwyer, M. B.; Avrunin, G. S.; & Corbett, J. C. "Patterns in Property Specifications for Finite-State Verification," 411-420. *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. Los Angeles, CA, May 16-22, 1999. New York, NY: Association for Computing Machinery (ACM), 1999.

[Gerth 01] Gerth, R. "Model Checking If Your Life Depends on It: A View from Intel's Trenches," [summary] 15. *Model Checking Software. 8th International SPIN Workshop. Proceedings* (in Lecture Notes in Computer Science [LNCS], volume 2057). Toronto, Ontario, Canada, May 19-20, 2001. Berlin, Germany: Springer-Verlag, 2001.

[Godefroid 97] Godefroid, P. A. "Model Checking for Programming Languages Using VeriSoft," 174-186. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGCAT Symposium on Principles of Programming Languages*. Paris, France, January 15-17, 1997. New York, NY: Association for Computing Machinery (ACM), 1997.

[Graf 97] Graf, S. & Saïdi, H. "Construction of Abstract State Graphs with PVS," 72-83. *Proceedings of the Computer Aided Verification. 9th International Conference (CAV '97)* (in Lecture Notes in Computer Science [LNCS], volume 1254). Haifa, Israel, June 22-25, 1997. Berlin, Germany: Springer-Verlag, 1997.

[Grumberg 94] Grumberg, O. & Long, D. E. "Model Checking and Modular Verification." *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994): 843-871.

[Havelund 00] Havelund, K.; Lowry, M.; Park, S.; Pecheur, C.; Penix, J.; Visser, W.; & White, J. "Formal Analysis of the Remote Agent Before and After Flight," 163-174. *LFM 2000: 5th NASA Langley Formal Methods Workshop (NASA/CP-2000-210100)*. Williamsburg, VA, June 13-15, 2000. Hampton, VA: NASA, 2000.

[Henzinger 00] Henzinger, T. A.; Qadeer, S.; & Rajamani, S. K. "Decomposing Refinement Proofs Using Assume-Guarantee Reasoning," 245-253. *Proceedings of International Conference on Computer-Aided Design*. San Jose, CA, November 5-9, 2000. New York, NY: IEEE Computing Society Press, 2000.

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

[Hoare 85] Hoare, C. A. R. *Communicating Sequential Processes*. London, UK: Prentice Hall, 1985.

[IBM 04] IBM. *Verification and Testing Technologies*. http://www.haifa.il.ibm.com/projects/verification/Formal_Methods-Home/ (July 2004).

[Ivers 03] Ivers, J. & Wallnau, K. "Preserving Real Concurrency." *Correctness of Model-Based Software Composition (CMD) Proceedings*, done in association with the *17th European Conference on Object-Oriented Programming* (in Lecture Notes in Computer Science [LNCS], volume 2743). Darmstadt, Germany, July 21-25, 2003. New York, NY: Springer-Verlag, 2003. http://www.ubka.uni-karlsruhe.de/vvv/ira/2003/13/13.pdf (2003).

[Kesten 00] Kesten, Y. & Pnueli, A. “Control and Data Abstraction: the Cornerstones of Formal Verification.” *International Journal on Software Tools for Technology Transfer 2*, 4 (March 2000): 328-342.

[Kurshan 94] Kurshan, R. P. *Computer-Aided Verification of Coordinating Processes: the Automata-Theoretic Approach*. Princeton, NJ: Princeton University Press, 1994.

[Lindsey 04] Lindsey, T. & Pecheur, C. “Simulation-Based Verification of Autonomous Controllers with Livingstone PathFinder,” 357-371. *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference (TACAS 2004)* held as part of the *Joint European Conferences on Theory and Practice of Software (ETAPS 2004)* (in Lecture Notes in Computer Science [LNCS], volume 2988). Barcelona, Spain, March 29-April 2, 2004. New York, NY: Springer-Verlag, 2004.

[Loiseaux 95] Loiseaux, C.; Graf, S.; Sifakis, J.; Bouajjani, A.; & Bensalem, S. “Property Preserving Abstractions for the Verification of Concurrent Systems.” *Formal Methods in System Design 6*, 1 (January 1995): 11-44.

[McMillan 97] McMillan, K. L. “A Compositional Rule for Hardware Design Refinement,” 24-35. *Proceedings of the Computer Aided Verification. 9th International Conference (CAV ’97)* (in Lecture Notes in Computer Science [LNCS], volume 1254). Haifa, Israel, June 22-25, 1997. Berlin, Germany: Springer-Verlag, 1997.

[McMillan 98] McMillan, K. L. “Verification of an Implementation of Tomasulo’s Algorithm by Compositional Model Checking,” 110-121. *Proceedings of the Computer Aided Verification: 10th International Conference (CAV ’98)* (in Lecture Notes in Computer Science [LNCS], volume 1427). Vancouver, BC, Canada, June 28-July 2, 1998. Berlin, Germany: Springer, 1998.

[Magee 01] Magee, J. & Kramer, J. *Concurrency: State Models & Java Programs*. West Sussex, England: Wiley Publication, 2001.

[Mellor 02] Mellor, S. J. & Balcer, M. J. *Executable UML: A Foundation for Model-Driven Architecture*. Boston, MA: Addison-Wesley, 2002.

[Misra 81] Misra, J. & Chandy, K. M. “Proofs of Networks of Processes.” *IEEE Transactions on Software Engineering 7*, 4 (July 1981): 417-426.

[Nelson 03] Nelson, S. & Pecheur, C. “Formal Verification of a Next-Generation Space Shuttle,” 53-67. *Formal Approaches to Agent-Based Systems: Second International Workshop (FAABS 2002)* (in Lecture Notes in Computer Science [LNCS], volume 2699). Greenbelt, MD, October 29-31, 2002. New York, NY: Springer-Verlag, 2003.

[OMG 02] Object Management Group. *Unified Modeling Language (UML), Version 1.5*. OMG document formal/2003-03-01. http://www.omg.org/technology/documents/formal/uml.htm (September 2002).

[Pnueli 77] Pnueli, A. “The Temporal Logic of Programs,” 46-57. *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*. Providence, RI, October 31-November 2, 1997. New York, NY: Institute of Electrical and Electronics Engineers, Inc. (IEEE), 1997.

[Pnueli 85] Pnueli, A. “Transition from Global to Modular Temporal Reasoning About Programs.”*Logics and Models of Concurrent Systems,* New York, NY: Springer-Verlag, 1985.

[Russinoff 98] Russinoff, D. “A Mechanically Checked Proof of IEEE Compliance of the Floating-Point Multiplication, Division, and Square Root Algorithms of the AMD-K7* Processor.” *London Mathematical Society Journal of Computation and Mathematics 1* (December 1998): 148-200.

[Stark 85] Stark, E. “A Proof Technique for Rely/Guarantee Properties,” 369-391. *Foundations of Software Technology and Theoretical Computer Science. Fifth Conference Proceedings.* New Dehli, India, December 16-18, 1985. Berlin, Germany: Springer-Verlag, 1985.

[Wallnau 03a] Wallnau, K. *Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC)* (CMU/SEI-2003-TR-009, ADA413574). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tr009.html

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

[Ziv 03] Ziv, A. "Cross-Product Functional Coverage Measurement with Temporal Properties-Based Assertions (Logic Verification)," 834-839. *Proceedings of the 6th Design, Automation, and Test in Europe (DATE 03).* Munich, Germany, March 3-7, 2003. Los Alamitos, CA: IEEE Computing Society, 2003.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	April 2004	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Overview of ComFoRT: A Model Checking Reasoning Framework	F19628-00-C-0003

6. AUTHOR(S)

James Ivers, Natasha Sharygina

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2004-TN-018

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

11. SUPPLEMENTARY NOTES

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

13. ABSTRACT (MAXIMUM 200 WORDS)

Component technologies are gaining acceptance in the software community as effective tools for quickly assembling increasingly complex systems from components. Most of the current component technologies, however, fail to help developers predict important software qualities like performance, safety, and reliability. A prediction-enabled component technology (PECT) augments the capabilities of a component technology with one or more reasoning frameworks that package quality specific analyses and the means to apply them to component-based systems. Model checking is an automated approach for exhaustively analyzing whether systems satisfy specific behavioral claims that can be used to characterize safety and reliability requirements. This technical note describes ComFoRT, a reasoning framework that packages the effectiveness of state-of-the-art model checking in a form that enables users to apply the analysis technique without being experts in its use, and its incorporation in a PECT.

14. SUBJECT TERMS	15. NUMBER OF PAGES
reasoning framework, model checking, software components, prediction-enabled component technology, formal verification, behavior analysis	54

16. PRICE CODE

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