The ComFoRT Reasoning Framework

Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt Wallnau

Software Engineering Institute, Carnegie Mellon University

1 Introduction

Model checking is a promising technology for verifying critical behavior of software. However, software model checking is hamstrung by scalability issues and is difficult for software engineers to use directly. The second challenge arises from the gap between model checking concepts and notations, and those used by engineers to develop large-scale systems. ComFoRT [15] addresses both of these challenges. It provides a model checker, Copper, that implements a suite of complementary complexity management techniques to address state space explosion. But ComFoRT is more than a model checker. The ComFoRT *reasoning framework* includes additional support for building systems in a particular component-based idiom. This addresses transition issues.

2 The Containerized Component Idiom

In the containerized component idiom, custom software is *deployed* into prefabricated containers. A component is a container and its custom code. Containers restrict visibility of custom code to its external environment (other components and a standard runtime environment), and vice versa. Components exhibit reactive behavior, characterized by how stimuli received through the container interface lead to responses emitted via the container interface. A runtime environment provides component coordination mechanisms (or "connectors") and implements other resource management policies (scheduling, synchronization, etc.). We define a component technology as an implementation of this design idiom [17], and many such implementations are possible [19]. Our approach has much in common with [12], although we give full behavioral models for components (UML statecharts and action language) and, subsequently, can generate full implementations of components and assemblies.

We formalize this idiom in the construction and composition language (CCL) [18]. The structural aspects of CCL (e.g., interfaces, hierarchy, topology) are similar to those found in a typical architecture description language [1]. The behavioral aspects of CCL use a subset of UML statecharts. Our formalization retains the statechart semantics already familiar to software engineers while refining it to precisely define those semantics intentionally left open in the standard. In formalizing both aspects of CCL, we exploit our connection with a specific component technology, which we use as the oracle for our choice of semantics.

ComFoRT exploits this design idiom and its formalization in several ways. Threading information in CCL specifications is exploited to generate concurrent state machines that more closely approximate actual concurrency than might otherwise be the case if threading were not considered [16]. The factoring of component-based systems into custom code and prefabricated containers and connectors presents opportunities for exploiting compositional reasoning. Models of containers and connectors can also be pre-fabricated; therefore, developers need only model their custom code to use the model checker. Moreover, as explained in Section 3, verification properties are specified using a formalism adapted to easily describe patterns of interaction among stateful components.

Because CCL is a design language, model checking can be used to verify early design decisions. However, model checking of software implementations is also possible because the model checker also processes a restricted form of ANSI-C source code (even though it is unsound with respect to pointers). The cumulative result is to make model checking more accessible to the practicing software engineer by using familiar notations, supporting verification throughout the development process, and providing automation to hide complexity.

3 Overview of the Model Checking Engine

Combined State Space Reduction. The ComFoRT model checker, Copper, is built on the top of the MAGIC tool [14]. Copper implements a number of state space reduction techniques, including 1) automated predicate abstraction, 2) counterexample-guided abstraction refinement (also known as a CEGAR loop), and 3) compositional reasoning. These techniques are widely used by the majority of software model checking tools (such as SLAM [2], BLAST [13], CBMC [9]). The advantage of Copper is that it combines *all* three of them in a complementary way to combat the state space explosion of software verification. For example, it enables compositional abstraction/refinement loop where each step of the CEGAR loop can be performed *one* concurrent unit at a time. Moreover, Copper integrates a number of complementary state space reduction techniques. An example is a *two-level abstraction approach* [3] where predicate abstraction for data is augmented by action-guided abstraction for events. Another key feature of the Copper approach is that if a property can be proved to hold or not based on a given finite set of predicates P, the predicate refinement procedure used in Copper automatically detects a minimal subset of P that is sufficient for the proof. This, along with the explicit use of compositionality, delays the onset of state-space explosion for as long as possible.

State/Event-based Verification. The Copper model checker provides formal models for software verification that leverage the distinction between data (states) and communication structures (events). Most formal models are either state-based (e.g., the Kripke structures used in model checking) or event-based (e.g., process algebras), but Copper provides models that incorporate both [7]. Semantically, this does not increase expressive power, since one can encode states as events or events as state changes, but providing both directly in the model fits more natural to software modeling and property specification. It is, indeed, essential in supporting the containerized component idiom. As importantly, it allows *more efficient verification* [7]. Copper models such systems as *Labeled Kripke Structures* and provides both state/event-LTL [7] and

ACTL formalisms [5]. Both versions of temporal logic are sufficiently expressive, yet allow a tractable implementation for model checking. The COPPER model checking algorithms support verification of both *safety* and *liveness* properties of state/event systems. Another feature of the state/event-based framework is a *compositional deadlock detection* technique [6] that not only efficiently detects deadlocks but also acts as an additional space reduction procedure.

Verification of Evolving Systems. The COPPER model checker also provides features that enable it to *automatically* verify *evolving software*. These features simplify verification throughout the development process—and through the extended life-cycle of a software system—by reducing the cost of re-verification when changes are made. We define verification of evolving systems as a component substitutability problem: (i) previously established properties must remain valid for the new version of a system, and (ii) the updated portion of the system must continue to provide all (and possibly more) services offered by its earlier counterpart. COPPER uses a completely automated procedure based on learning techniques for regular sets to solve the substitutability problem in the context of verifying individual component upgrades [4]. Furthermore, COPPER also supports analysis of component substitutability in the presence of *simultaneous upgrades of multiple components* [8]. COPPER uses *dynamic* assume-guarantee reasoning, where previously generated assumptions are reused and altered on-the-fly to prove or disprove the global safety properties on the updated system.

4 Tool Support

COMFORT consists of two sets of tools: those for generating the state machines to be verified and those that perform the actual model checking. The first set deals with the topics discussed in Section 2, parsing and performing semantic analysis of design specifications (in CCL) and the generation of the state machines in the input language of COPPER. COPPER, as discussed in Section 3, is the model checker at the core of COMFORT. COPPER was built on top of MAGIC, portions of which we developed together with collaborators from Carnegie Mellon's School of Computer Science specifically to support COMFORT[1]. COPPER has since evolved beyond the MAGIC v.1.0 code base, and a brief overview of some of the key features of COPPER and their lineage in terms of various tool releases is found in Figure 1. COMFORT is available at http://www.sei.cmu.edu/pacc/comfort.html.

5 Results

We have used COMFORT to analyze several industrial benchmarks. Our first benchmark was derived from the OpenSSL-0.9.6c implementation of SSL. Specifically, we verified the implementation of the handshake between a client (2500 LOC) and a server (2500 LOC) attempting to establish a secure connection with respect to several properties derived from the SSL specification. Figure 2

[1] The reader, therefore, should not be confused by the fact that results of this collaboration have been presented in the contexts of both projects.

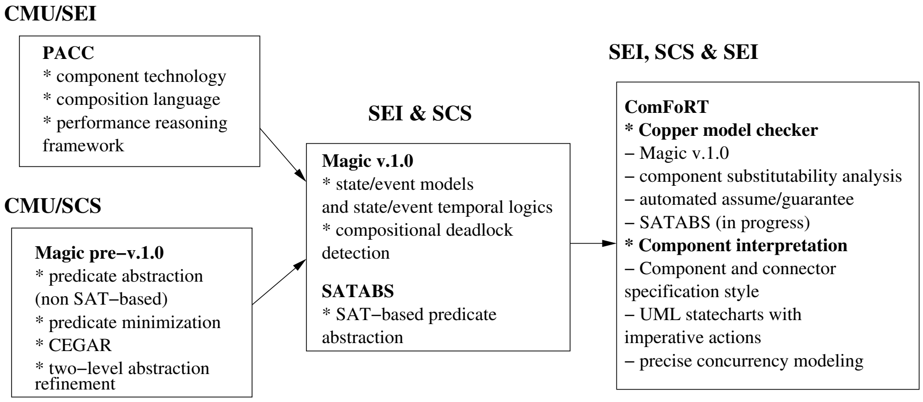

Fig. 1. Evolution of the MAGIC and ComFoRT projects

shows verification results of two properties, each of which was expressed using only states (`ss` suffix) and both states and events (`se` suffix). Note that the models depend on the property - and hence are different for the pure-state and state/event versions, even though they are constructed from the same source code. As shown in Figure 2, verification of the state/event properties outperforms the corresponding pure-state properties.

Name	St(B)	Tr(B)	St(Mdl)	T(BA)	T(Mdl)	T(Ver)	T(Total)	Mem
ssl-1-ss	25	47	25119360	1187	69969	*	*	324
ssl-1-se	20	45	13839168	848	37681	113704	153356	165
ssl-2-ss	25	47	33199244	1199	67419	3545288	3615016	216
ssl-2-se	18	40	16246624	814	38080	298601	338601	172

Fig. 2. **St(B)** and **Tr(B)** = number of Büchi states and transitions; **St(Mdl)** = number of model states; **T(Mdl)** = model construction time; **T(BA)** = Büchi construction time; **T(Ver)** = model checking time; **T(Total)** = total verification time. Times are in milliseconds. **Mem** = memory in MB. A * $\equiv$ model checking aborted after 2 hours.

Two other benchmarks we have used are Micro-C OS and the interprocess-communication library of an industrial robot controller. With Micro-C OS, verification of source code revealed a locking protocol violation. With the communication library, verification of CCL models derived from the implementation revealed a problem wherein messages could be misrouted. In both cases, the respective developers informed us that the problems had been detected and fixed; in the latter case, the problem was undetected during seven years of testing.

6 Future Work

We are currently working on a number of additions to ComFoRT. One is the incorporation into Copper a SAT-based predicate abstraction technique [10] that eliminates the exponential number of theorem prover calls of the current abstraction procedure. Another is the use of a simpler language for expressing verification properties, such as a pattern language [11]. A third is a technique for confirming that design-level (i.e., CCL designs) verification results are satisfied by eventual component implementations by proving a conformance relation between the model and its implementation.

References

1. F. Achermann, M. Lumpe, J. Schneider, and O. Nierstrasz. Piccola – a Small Composition Language. In *Formal Methods for Distributed Processing–A Survey of Object-Oriented Approaches*. 2002.
2. T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.
3. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *FMSD*, 25(2), 2004.
4. S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software. In *SAVCBS'04: Worksh. on Specification and Verification of Component-based Systems*, 2004.
5. E. Clarke, S. Chaki, O. Grumberg, T. Touili, J. Ouaknine, N. Sharygina, and H. Veith. An expressive verification framework for state/event systems. Technical Report CS-2004-145, CMU, 2004.
6. E. Clarke, S. Chaki, J. Ouaknine, and N. Sharygina. Automated, compositional and iterative deadlock detection. In *2nd ACM-IEEE MEMOCODE 04*, 2004.
7. E. Clarke, S. Chaki, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *IFM 04: Integrated Formal Methods*, LNCS 2999, 2004.
8. E. Clarke, S. Chaki, N. Sharygina, and N. Sinha. Dynamic component substitutability analysis. In *FM 2005: Formal Methods, to appear*. LNCS, 2005.
9. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
10. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2), 2004.
11. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st ICSE*, 1999.
12. J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *ICSE*, pages 160–173, 2003.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages 02*.
14. http://www.cs.cmu.edu/chaki/magic. Magic tool.
15. J. Ivers and N. Sharygina. Overview of ComFoRT: A Model Checking Reasoning Framework. Technical Report CMU/SEI-2004-TN-018, SEI, CMU, 2004.
16. J. Ivers and K. Wallnau. Preserving real concurrency. In *Correctness of model-based software composition Workshop*, July 2003.
17. K. Wallnau. Vol III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical Report CMU/SEI-2003-TR-009, SEI,CMU, 2003.
18. K. Wallnau and J. Ivers. Snapshot of CCL: A Language for Predictable Assembly. Technical Report CMU/SEI-2002-TR-031, SEI, CMU, 2002.
19. N. Ward-Dutton. Containers: A sign components are growing up. *Application Development Trends*, pages 41–44,46, January 2000.