Correctness of Model-based Component Composition without State Explosion*

Paul C. Attie David H. Lorenz

Northeastern University
College of Computer & Information Science
Boston, Massachusetts 02115 USA
{attie,lorenz}@ccs.neu.edu

Abstract. We present a methodology for designing component-based systems and verifying their temporal behavior properties. Our verification method is mostly automatic, and is not susceptible to the well-known *state-explosion* problem, which has hitherto severely limited the practical applicability of automatic verification methods. Our method specifies the externally visible behavior of each component C as several *behavioral interface automaton* (BIA), one for each of the other components which C interacts directly with. A BIA is a finite-state automaton whose transitions can be labeled with method calls. For each pair of directly interacting components, we compute the product of the BIA. These "pair machines" are then verified mechanically. The verified "pair properties" are then combined deductively to deduce global properties. Since the pair-machines are the product of only two components, they are small, and so their mechanical verification, e.g., by model checking, does not run up against state-explosion. The use of several BIA per component enables a clean separation between interfaces, so that the interactions of a component C with several other components are cleanly separated, and can be inspected in isolation. This in itself promotes the understandability of a design. Our method also enhances extensibility. If a component is modified, only the pairs in which that component is involved are affected. The rest of the system is undisturbed. To our knowledge, our method is the first approach to behavioral compatibility that does not suffer from state-explosion.

1 Introduction

Software components [17] are supposed to make software less fragile and more reliable. In practice, however, part of the fragility is merely shifted from the component artifacts to the connectors and the composition process. When the composition is unreliable, component systems are just as fragile and unreliable as monolithic software. Improving the theoretical and practical foundation of third-party composition techniques [21] is thus essential to improving overall component software reliability.

In this paper, we make initial steps toward a new component model which supports behavioral interoperability and is based on the use of temporal logic and automata to

* This work was supported in part by the National Science Foundation (NSF) under Grant No. CCR-0204432, and by the Institute for Complex Scientific Software (www.icss.neu.edu) at Northeastern University. An extended version of this paper is available as [3].

specify and reason about concurrent component systems. Unlike other temporal logic and automata-based methods for software components, our work avoids using exhaustive state-space enumeration, which quickly runs up against the *state-explosion* problem where the number of global states of a system is exponential in the number of its components.

We present formal analysis and synthesis techniques that address issues of behavioral compatibility amongst components, and enable reasoning about the global behavior (including temporal behavior, i.e., safety and liveness) of an assembly of components. By avoiding state-explosion, our technique is not restricted to small, unrealistic applications.

1.1 Component interoperability

Components are "units of independent production, acquisition, and deployment" [17]. In *component-based software engineering* (CBSE) [10], software development is decoupled from assembly and deployment. Third party composition (assembly) is the activity of connecting components, which originate from different third party component providers in binary format and without their source code. During assembly, the application (or component) is assembled from other (compiled) components. The activity takes place after the compilation of the components and before the deployment of the application (which might be itself a compound component).

For two components, which were independently developed, to be deployed and work together, third-party composition must allow the flexibility of assembling even dissimilar, heterogeneous, precompiled third-party components. In achieving this flexibility, a delicate balance is preserved between prohibiting the connecting of incompatible components (avoiding false positive), while permitting the connecting of "almost compatible" components through adaptation (avoiding false negative). This is achieved during assembly through introspection, compatibility checks, and adaptability.

1.2 Interface compatibility

Parnas's principles [16] of information hiding for modules emphasize the separation of interface from implementation: components providing different implementations of the same interface can be swapped without having a functional affect on clients; two components need to agree on the interface in order to communicate. This works well in object-oriented programming where the design is centralized, but is not practical in component-based designs [14]. Agreement beforehand is possible only if third-party component providers were coordinated.

Extending Parnas's principles to component-based programming (CBP), the component clients (i.e., other components) must be provided with composition information and nothing more. Even agreement on the interface is no longer an accepted level of exported information. Components gathered from third parties are unlikely and cannot be expected to agree on interfaces beforehand. For third-party composition to work, components need to agree on how to agree rather then agree on the interface.

Indeed, CBP builder environments typically apply two mechanisms to overcome this difficulty and support third-party composition [13]. First, to check for interface

compatibility, builders use *introspection*. Introspection is a means for discovering the component interface. Second, builders support *adaptability* by generating adapters to overcome differences in the interface. Adapters are a means of fixing small mismatches when the interfaces are not syntactically identical.

1.3 Behavioral compatibility

The goal of work in behavioral compatibility for components is to develop support in CBP for *behavioral introspection* and *behavioral adaptability* that can be scaled up for constructing large complex component systems. While there is progress in addressing behavioral introspection and adaptability [22, 20, 23, 19, 18] there is no progress in dealing with the state explosion problem. The main focus of this work is in addressing the latter in a manner that can be applied to event-based components.

Currently, the introspector reveals only the interface, and adapters are used in an ad-hoc manner relying on names and types only. There are emerging proposals for handling richer interface mechanisms that express contractible constraints on the interface, e.g., the order in which the functions should be called, or the result of a sequence of calls. These methods typically rely on defining finite-state "behavioral" automata that express state changes. When two components are connected, the two automata can be tested for compatibility by producing their automata-theoretic product. This fails to provide a practical foundation for software growth, because of sate explosion; computation of the product of K behavioral automata, each with $O(N)$ states, generates a product automaton of size $O(N^K)$. We address the challenge of avoiding state explosion. Elsewhere [3, 4] we present a pairwise design of an elevator system which, when scaled up to 200 floors, requires an upper bound of only $1,166,400$ states, instead of the 10^{180} that an approach which computes the product of all components would require. This is well within the reach of current model checkers.

2 Formal methods for components and composition correctness

Our interest is in large systems of concurrently executing components. A crucial aspect of the correctness of such systems is their temporal behavior. Behavioral properties can be classified as follows [12]: (1) *Safety properties*: "nothing bad happens" — for example, when an elevator is moving up, it does not attempt to move down without stopping first, and (2) *Liveness properties*: "progress occurs in the system" — for example, if a button inside an elevator is pressed, then the elevator eventually arrives at the corresponding floor. The required behavioral properties are given by a *specification*, which precisely documents what the system must achieve. *Formal methods* are those that provide a rigorous mathematical guarantee that a large software system conforms to a specification. Formal methods can be roughly classified as (1) *Proof-theoretic*: a suitable deductive system is used, and correctness proofs are built manually, or using a theorem prover, and (2) *Model-theoretic*: a model of the run-time behavior of the software is built, and this model is checked (usually mechanically) for the required properties. In our work, we emphasize model-theoretic methods, due to their greater potential for automation.

	Interface compatibility	Automaton (BIA) compatibility	Behavioral compatibility
Export	interface	interface + automaton	complete code
Reuse	black box	adjustable	white box
Encapsulation	highest	adjustable	lowest
Interoperability	unsafe	adjustable	safe
time complexity	linear	polynomial for finite state	undecidable
Assembly properties	none	provable from pair properties	complete but impractical
Assembly behavior	none	synthesizable from pairwise behavior	complete but impractical

Table 1. The interoperability space for components

3 The interoperability space for components

A *behavioral interface automaton* (BIA) of a component expresses some aspects of that components run-time (i.e., temporal) behavior. Depending on how much information about temporal behavior is included in the automaton, there is a spectrum of state information ranging from a maximal BIA for the component (which includes every transition the component makes, even internal ones), to a trivial automaton consisting of a single state. Thus, any BIA for a component can be regarded as a homomorphic image of the maximal automaton. This spectrum refines the traditional white-box/black-box spectrum of component reuse, ranging from exporting the complete source code (maximal automaton) of the component—white-box, and exporting just the interface (trivial automaton)—black box. Table 1 displays this spectrum.

In practice, it is unrealistic to expect the programmer to provide the maximal BIA, just as precisely specified semantics are rarely part of programming practices. As long as the most important behavioral properties (e.g., the safety-critical ones) can be expressed and established, a homomorphic image of the maximal automaton (which omits some information on the components behavior) is sufficient (Table 1 middle column).

The BIA can be provided by the component designer and verified by the compiler (just like typed interfaces are) using techniques such as abstraction mappings and model checking. Verification is necessary to ensure the correctness of the BIA, i.e., that it is truly a homomorphic image of the maximal automaton. Alternatively, the component compiler can generate a BIA from the code, using, for example, abstract interpretation or machine learning [15]. In this case, the BIA will be correct by construction. We assume the first option for third party components, and will explore the second option for components assembled in our builder.

4 Avoiding state-explosion by pairwise composition

In [1, 2], we present a method for the synthesis of finite-state concurrent programs from specifications expressed in the branching-time propositional temporal logic CTL [8].

This method avoids exhaustive state-space search. Rather than deal with the behavior of the program as a whole, the method instead generates the interactions between processes *one pair at a time*. Thus, for every pair of processes that interact, a *pair-machine* is constructed that gives their interaction. Since the pair-machines are small ($O(N^2)$), they can be built using exhaustive methods. A *pair-program* can then be extracted from the pair-machine. This extraction operation takes every transition of the pair-machine and realizes it as a piece of code in the pair-program [2, 9]. The final synthesized program is generated by a syntactic composition of all the pair-programs. This composition has a conjunctive nature: a process P_i can execute a transition if and only if that transition is permitted by *every* pair-program in which P_i participates. Thus, two pair-programs which have no processes in common do not interact directly. Two pair-programs that do have a process P_i in common will interact via P_i: the pair-programs in effect must synchronize whenever P_i makes a transition, so that the transition is executed in both pair-programs simultaneously. For example, if $P_1 \parallel P_2$, $P_2 \parallel P_3$, and $P_1 \parallel P_3$ are three pair-programs which all implement a two-process mutual exclusion algorithm, then they can be composed as discussed above into a single program which implements three-process mutual exclusion. In this program, when P_1 wishes to access the critical section, it must be permitted to do so by both P_2 (as per the $P_1 \parallel P_2$ pair-program) and by P_3 (as per the $P_1 \parallel P_3$ pair-program).

Due to the complexity of the synthesis and verification problems for finite-state concurrent programs, any efficient synthesis method is necessarily *incomplete*: it may fail to produce a program that satisfies a given specification even though such a program exists. In the synthesis method of [1, 2], the incompleteness takes the form of two technical assumptions that the pair-programs must satisfy in order for the synthesized program to be correct. One technical assumption requires that after a process P_i executes, either it can execute again (i.e., is enabled), or it does not block any other process. This prevents deadlock. The other technical assumption requires that a process cannot forever block another process if the second process must make progress in order to satisfy a liveness property in the specification. This guarantees liveness.

We refer the reader to [1, 2] for examples of synthesis of solutions to the following problems, all for an arbitrarily large number of processes: n-process mutual exclusion, dining philosophers, drinking philosophers [5], k-out-of-n mutual exclusion, and two-phase commit.

4.1 Applying pairwise composition to component assembly

To apply the pairwise method to components, we must be able to define the pairwise interaction amongst components. We do this by extending the component model so that each component C is accompanied by several BIA [7, 20], one for each of the other components that C interacts directly with. The BIA provides information about the externally observable temporal behavior of the component. For example, such an automaton could provide information on the order in which a component makes certain method calls to other components.

Given two components and their BIA, we construct the pair-machine for their interaction by simply taking the automata-theoretic product of the BIA. We can then model

check the pair-machine for the desired behavioral compatibility among the two components. If successful, we can then use this pair-machine as input to the pairwise method, as discussed above.

4.2 Discussion

The pairwise architecture enables a clean separation between interfaces. In the usual approach, a component has a single interface, through which it interacts with all other components. Thus, different interactions with different components are all mediated through the same interface. This results in an "entangling" of the run-time behaviors of various components, and makes reasoning (both mechanical and manual) about the temporal behavior of a system difficult. By contrast, our architecture "disentangles" the interactions of the components, so that the interaction of two components is mediated by a pair of interfaces, one in each component, that are designed expressly for only that purpose, and which are not involved in any other interaction. Thus, our architecture provides a clean separation of the run-time interaction behaviors of the various component-pairs. This simplifies both mechanical and manual reasoning, and results in a design and verification methodology that scales up.

Our architecture also facilitates extensibility: if a new component is added to the system, all that is required is to design new interfaces for interaction with that component. The interfaces between all pre-existing pairs of components need not be modified. Furthermore, all verification already performed of the behavior of pre-existing component-pairs does not need to be redone. Thus, both design and verification are extensible in our methodology. We can also apply our approach at varying degrees of granularity, depending on how much functionality is built into each component.

Vanderperren and Wydaeghe [22, 20, 23, 19, 18] have developed a component composition tool (PascoWire) for JavaBeans that employs automata-theoretic techniques to verify behavioral automata. They acknowledge that the practicality of their method is limited by state-explosion. Incorporating our technique with their system is an avenue for future work.

DeAlfaro and Henzinger [7] have defined a notion of interface automaton, and have developed a method for mechanically verifying temporal behavior properties of component-based systems expressed in their formalism. Unfortunately, their method computes the automata-theoretic product of all of the interface automata in the system, and is thus subject to state-explosion.

5 Conclusion

We have presented a methodology for designing components so that they can be composed in a pairwise manner, and their temporal behavior properties verified without state-explosion. Our method specifies the externally visible behavior of each component C as several behavioral interface automaton, one for each of the other components which C interacts directly with. Finite-state automata are widely used as a specification formalism, and so our work in compatible with the mainstream of component-based software engineering.

Ensuring the correct behavior of large complex systems is the key challenge of software engineering. Due to the ineffectiveness of testing, formal verification has been regarded as a possible approach, but has been problematic due to the expense of carrying out large proofs by hand, or with the aid of theorem provers. One proposed approach to making formal methods economical is that of automatic model checking [6]: the state space of the system is mechanically generated and then exhaustively explored to verify the desired behavioral properties. Unfortunately, the number of global states is exponential in the number of components. This state-explosion problem is the main impediment to the successful application of automatic methods such as model-checking and reachability analysis. Our approach is a promising direction in overcoming state-explosion. In addition to the elevator problem, the pairwise approach has been applied successfully to the two-phase commit problem [1] and the dining and drinking philosophers problems [2].

Large scale component-based systems are widely acknowledged as a promising approach to constructing large-scale complex software systems. A key requirement of an successful methodology for assembling such systems is to ensure the behavioral compatibility of the components with each other. This paper presents a first step towards a practical method for achieving this.

References

1. P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *CONCUR'99: 10th International Conference on Concurrency Theory*, number 1664 in LNCS. Springer-Verlag, Aug. 1999.
2. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, Jan. 1998.
3. P. C. Attie and D. H. Lorenz. Establishing behavioral compatibility of software components without state explosion. Technical Report NU-CCIS-03-02, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Mar. 2003. `http://www.ccs.neu.edu/home/lorenz/papers/reports/NU-CCIS-03-02.html`.
4. O. Aytar, P. C. Attie, and D. H. Lorenz. An implementation of an elevator system in the ioa language and toolset. Technical Report NU-CCIS-03-04, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Mar. 2003. `http://www.ccs.neu.edu/home/lorenz/papers/reports/NU-CCIS-03-04.html`.
5. K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Reading, Mass., 1988.
6. E. M. Clarke, E. A. Emerson, and P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.
7. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2001.
8. E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Models and Semantics*. The MIT Press/Elsevier, Cambridge, Mass., 1990.
9. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
10. G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

11. ICSE 2001. *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, May 12-19 2001. IEEE Computer Society.
12. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, Mar. 1977.
13. D. H. Lorenz and P. Petkovic. Design-time assembly of runtime containment components. In Q. Li, D. Firesmith, R. Riehle, G. Pour, and B. Meyer, editors, *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems*, pages 195–204, Santa Barbara, CA, July 30-Aug. 4 2000. TOOLS 34 USA Conference, IEEE Computer Society.
14. D. H. Lorenz and J. Vlissides. Designing components versus objects: A transformational approach. In ICSE 2001 [11], pages 253–262.
15. E. Mäkinen and T. Systä. MAS - an interactive synthesizer to support behavioral modeling in uml. In ICSE 2001 [11], pages 15–24.
16. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communication of the ACM*, 15(12):1059–1062, 1972.
17. C. Szyperski. *Component-Oriented Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
18. W. Vanderperren. A pattern based approach to separate tangled concerns in component based development. In Y. Coady, editor, *Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 71–75, Enschede, The Netherlands, Apr. 2002.
19. W. Vanderperren and B. Wydaeghe. Separating concerns in a high-level component-based context. In EasyComp Workshop at ETAPS 2002, April 2002.
20. W. Vanderperren and B. Wydaeghe. Towards a new component composition process. In Proceedings of ECBS 2001, April 2001.
21. K. C. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. Software Engineering. Addison-Wesley, 2001.
22. B. Wydaeghe. PACOSUITE: Component composition based on composition patterns and usage scenarios. PhD Thesis.
23. B. Wydaeghe and W. Vandeperren. Visual component composition using composition patterns. In Proceedings of Tools 2001, July 2001.