Behavioral Compatibility without State Explosion: Design and Verification of a Component-based Elevator Control System*

Paul C. Attie[1], David H. Lorenz[2], Aleksandra Portnova[3], and Hana Chockler[4]

[1] American University of Beirut, Beirut, Lebanon. paul.attie@aub.edu.lb
[2] University of Virginia, Charlottesville, VA 22904, USA. lorenz@cs.virginia.edu
[3] Northeastern University, Boston, MA 02115, USA. portnova@ccs.neu.edu
[4] WPI, Worcester, MA 01609, USA. hanac@theory.csail.mit.edu

Abstract. Most methods for designing component-based systems and verifying their compatibility address only the syntactic compatibility of components; no analysis of run-time behavior is made. Those methods that do address run-time behavior suffer from *state-explosion*: the exponential increase of the number of global states, and hence the complexity of the analysis, with the number of components. We present a method for designing component-based systems and verifying their behavioral compatibility and temporal behavior that is not susceptible to state explosion. Our method is mostly automatic, with little manual deduction required, and does not analyze a large system of connected components at once, but instead analyzes components two-at-a-time. This pair-wise approach enables the automatic verification of temporal behavior, using model-checking, in time polynomial in the number and size of all components. Our method checks that behavior of a pair of interacting components conforms to given properties, specified in temporal logic. Interaction of the components is captured in a product of their behavioral automata, which are provided as a part of each component's interface. We demonstrate the effectiveness of our method by applying it to the design and verification of a component-based elevator control algorithm.

1 Introduction

Monolithic software systems are fragile and unreliable. *Component-based software engineering* (CBSE) [34, 38, 19] alleviates this inherent software problem. Third-party composition of software systems, comprising reliable components from trustworthy third-party providers, reduces the system's overall fragility [28]. 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 is thus essential to improving overall component software reliability.

* This work was supported in part by NSF's Science of Design program under Grants Number CCF-0438971 and CCF-0609612.

In this paper, we lay a foundation for a new component model which supports behavioral interoperability and is based on the use of temporal logic and automata to 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*: the number of global states of a system is exponential in the number of its components. We present formal analysis and synthesis techniques that addresses issues of behavioral compatibility among components, and enables reasoning about global behavior (including temporal behavior, i.e., safety and liveness) of an assembly of components.

We illustrate the model concretely by means of an example design for an elevator system, which can scale up in size (number of floor and button components) and still be model-checked. Designing a component-based elevator system that can be scaled up is a canonical Software Engineering problem since it runs up against state-explosion. Our methodology, however, permits model-checking in time polynomial in the number and size of components.

2 Problem and Approach in a Nutshell

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 components. In achieving this flexibility, a delicate balance is preserved between prohibiting the connecting of incompatible components (avoiding false positives), while permitting the connecting of "almost compatible" components through adaptation (avoiding false negatives). This is achieved during assembly through introspection, compatibility checks, and adaptability.

CBSE builder environments typically apply two mechanisms to support third-party composition. First, to check for *interface compatibility*, builders use *introspection*. Introspection is a means of 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.

The goal in *behavioral compatibility* for components is to develop support in CBSE 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 [40, 35, 39, 36, 37] there is little progress in dealing with the *state explosion problem*.

2.1 The state explosion problem

Many current mechanical methods for reasoning about behavior (of finite state systems) generally rely on some form of exhaustive state-space search to generate all the possible behaviors. These methods are thus susceptible to state explosion: the number of global states of a concurrent system consisting of n components, each with $O(l)$ local states, is in $O(l^n)$. Approaches to dealing with state explosion include compositional verification [29, 18, 13, 9, 8, 25] (and the

strongly related assume-guarantee reasoning [1, 20]), abstraction [30, 12, 23, 24], and symmetry reduction [15, 16, 11, 10].

Current methods typically rely on defining finite-state "behavioral" automata that express state changes. The automata-theoretic product of the behavioral automata of two components will then describe the resulting behavior when these two components are connected. Thus, the two components can be checked for compatibility by model checking this product. When a third component is subsequently connected to the first two, one then needs to generate the product of all three behavioral automata. Thus, this fails to provide a practical method for checking large systems, since taking the product of n automata incurs state explosion.

2.2 Avoiding state-explosion by pair-wise composition

To overcome state-explosion, we eschew the computation of the product of all n behavioral automata. Instead, we compute the products of *pairs* of behavioral automata, corresponding to the pairs of components that interact directly.[5] In the worst case, where all components interact (where the largest component has $O(l)$ local states), this has complexity $O(n^2 l^2)$. This low polynomial complexity means that our method scales up to large systems. We verify temporal behavior "pair-properties" of these "pair-products." These give us properties of the interactions of all component-pairs, when considered in isolation. We then combine such "pair-properties" to deduce global properties of the entire system by means of temporal logic deductive systems [17]. Since the pair-properties embody the complexity of the component interaction, this deductive part of the verification is quite short.

Our approach involves abstraction in going from a component to its behavioral automaton. It applies even when all components are functionally different, and so is not a form of symmetry reduction. Our approach combines pair-properties verified of each pair-product to deduce the required global properties. Each pair-product represents two components *interacting in isolation*. Our approach therefore does not involve the usual "assume guarantee" proof rule typical of compositional approaches, where each component is verified correct using the assumption that the other components are correct, with due care taken to avoid cyclic reasoning.

The main insight of this paper is that components can be designed to enable this pair-wise verification, thus supporting behavioral compatibility checks that scale up to large complex systems [7].

3 Technical Preliminaries

I/O automata We augment the standard definition of I/O automata [31] to accommodate propositional labelings of states. An *augmented input/output (I/O) automaton* A is a tuple

$$\langle\!\langle states(A), start(A), sig(A), steps(A), prop(A), label(A) \rangle\!\rangle$$

[5] For clarity, we assume all connectors involve exactly two components. The methodology can be easily generalized to verify connectors between multiple components.

as follows. $states(A)$ is a set of states; $start(A) \subseteq states(A)$ is a nonempty set of start states; $sig(A) = (in(A), out(A), int(A))$ is an action signature, where $in(A)$, $out(A)$ and $int(A)$ are pair-wise disjoint sets of input, output, and internal actions, respectively, (let $acts(A) = in(A) \cup out(A) \cup int(A)$); $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ is a transition relation; $prop(A)$ is a set of atomic propositions; and $label(A) : states(A) \mapsto 2^{prop(A)}$ is a labeling function. If $states(A)$, $acts(A)$ and $prop(A)$ are all finite, then A is a *finite-state* I/O automaton. $label(A)(s)$ gives the atomic propositions that are true in state s.

Let $s, s', u, u', \ldots$ range over states and $a, b, \ldots$ range over actions. Write $s \xrightarrow{a}_A s'$ iff $(s, a, s') \in steps(A)$. We say that a is *enabled* in s. Otherwise a is *disabled* in s. I/O automata are required to be *input enabled*: every input action is enabled in every state. An *execution fragment* α of automaton A is an alternating sequence of states and actions $s_0 a_1 s_1 a_2 s_2 \ldots$ such that $(s_i, a_{i+1}, s_{i+1}) \in steps(A)$ for all $i \geq 0$, i.e., α conforms to the transition relation of A. Furthermore, if α is finite then it ends in a state. An *execution* of A is an execution fragment that begins with a state in $start(A)$. $execs(A)$ is the set of all executions of A. A state of A is *reachable* iff it occurs in some execution of A. Two I/O automata are *compatible* iff they have no output actions and no atomic propositions in common, and no internal action of one is an action of the other. A set of I/O automata is compatible iff every pair of automata in the set is compatible.

Definition 1 (Parallel Composition of I/O automata). *Let $A_1, \ldots, A_n$, be compatible I/O Automata. Then $A = A_1 \parallel \cdots \parallel A_n$ is the I/O automaton*[6] *defined as follows. $states(A) = states(A_1) \times \cdots \times states(A_n)$; $start(A) = start(A_1) \times \cdots \times start(A_n)$; $sig(A) = (in(A), out(A), int(A))$ where $out(A) = \bigcup_{1 \leq i \leq n} out(A_i)$, $in(A) = \bigcup_{1 \leq i \leq n} in(A_i) - out(A)$, $int(A) = \bigcup_{1 \leq i \leq n} int(A_i)$; $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ consists of all the triples $(\langle s_1, \ldots, s_n \rangle, a, \langle t_1, \ldots, t_n \rangle)$ such that $\forall i \in \{1, \ldots, n\}$: if $a \in acts(A_i)$, then $(s_i, a, t_i) \in steps(A_i)$, otherwise $s_i = t_i$; $prop(A) = \bigcup_{1 \leq i \leq n} prop(A_i)$; and $label(A)(\langle s_1, \ldots, s_n \rangle) = \bigcup_{1 \leq i \leq n} label(A_i)(s_i)$.*

Let $A = A_1 \parallel \cdots \parallel A_n$ be a parallel composition of n I/O automata. Let s be a state of A. Then $s \restriction A_i$ denotes the i'th component of s, i.e., the component of s that gives the local state of A_i, $i \in \{1, \ldots, n\}$. Let $\varphi = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$. Then $s \restriction A_\varphi$ denotes the tuple $\langle s_j \restriction A_{i_1}, \ldots s_j \restriction A_{i_m} \rangle$. A *subsystem* of A is a parallel composition $A_{i_1} \parallel \cdots \parallel A_{i_m}$, where $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$. We define the projection of an execution of A onto a subsystem of A in the usual way: the state components for all automata other than $A_{i_1}, \ldots, A_{i_m}$ are removed, and so are all actions in which none of the $A_{i_1}, \ldots, A_{i_m}$ participate:

Definition 2 (Execution projection). *Let $A = A_1 \parallel \cdots \parallel A_n$ be an I/O automaton. Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots s_{j-1} a_j s_j \ldots$ be an execution of A. Let $\varphi = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, and let $A_\varphi = A_{i_1} \parallel \cdots \parallel A_{i_m}$. We define $\alpha \restriction A_\varphi$ as the sequence resulting from removing all $a_j s_j$ such that $a_j \notin acts(A_\varphi)$ and replacing each s_j by $s_j \restriction A_\varphi$.*

[6] Formally, A is a state-machine. It is easy to show, though, that A is in fact an I/O automaton.

Proposition 1 (Execution projection). *Let $A = A_1 \parallel \cdots \parallel A_n$ be an I/O automaton. Let $\alpha \in execs(A)$. Let $\varphi = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, and let $A_\varphi = A_{i_1} \parallel \cdots \parallel A_{i_m}$. Then $\alpha \lceil A_\varphi \in execs(A_\varphi)$.*

Proof. Immediate from the standard execution projection result for I/O automata [31], when considering the subsystem A_φ as a single I/O automaton.

4 Formal Methods for Composition Correctness

Attie and Emerson [4, 5] present a temporal logic synthesis method for shared memory programs that 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 ($O(l^2)$), they can be built using exhaustive methods. A *pair-program* can then be extracted from the pair-machine. The final program is generated by a syntactic composition of all the pair-programs.

Here, we extend this method to the I/O automaton [31] model, which is *event-based*. Unlike [4, 5], which imposed syntactic restrictions (variables must be shared pairwise), the method presented here can be applied to any component-based system expressed in the I/O automaton notation. It is straightforward to extend the results presented here to any event-based formalism with a well-defined notion of composition.

The method of [4] is *synthetic*: for each interacting pair, the problem specification gives a formula that specifies their interaction, and that is used to synthesize the corresponding pair-machine. We also consider the *analytic* use of the pair-wise method: if a program is given, e.g., by manual design, then generate the pair-machine by taking the concurrent composition of the components one pair at a time. The pair-machines can then be model-checked for the required conformance to the specification. If the pair-machines behave as required, then we can deduce that the overall program is correct.

In our method, the desired safety and liveness properties are automatically verified (e.g., by model checking) in pair systems and the correctness of the whole system deduced from the correctness of these pair systems. We start with formally proving the propagation of safety and liveness properties from pair systems to the large system. We use propositional linear-time temporal logic [33, 17] without the nexttime modality (LTL-X), and with weak action fairness, to specify properties. LTL-X formulae are built up from atomic propositions, boolean connectives, and the temporal modality U (strong until). LTL-X semantics is given by the $\models$ relation, which is defined by induction on LTL-X formula structure. Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ be an infinite execution fragment of A, $\alpha^i = s_i a_{i+1} s_{i+1} a_{i+2} s_{i+2} \ldots$, a suffix of α, and p be an atomic proposition. Then $A, \alpha \models p$ iff $p \in label(A)(s_0)$, and $A, \alpha \models f \mathsf{U} g$ iff $\exists i \geq 0 : A, \alpha^i \models g$ and $\forall j \in \{0, \ldots, i-1\} : A, \alpha^j \models f$. We define the abbreviations $\mathsf{F} f = \mathbf{true} \mathsf{U} f$ ("eventually"), $\mathsf{G} f = \neg \mathsf{F} \neg f$ ("always"), $f \mathsf{U}_\mathsf{w} g = (f \mathsf{U} g) \vee \mathsf{G} f$ ("weak until"), and $\overset{\infty}{\mathsf{F}} f = \mathsf{GF} f$ ("infinitely often").

Fairness constraints allow us to filter away irrelevant executions. We use weak action fairness: an execution fragment α of A is fair iff it is infinite and every action of A is either infinitely often executed along α or infinitely often disabled along α. Define $A, s \models_\Phi f$ iff f holds along all fair execution fragments of A starting in s, and $A \models f$ iff f holds along all fair executions of A.

Let $A = A_1 \parallel \cdots \parallel A_n$ be the large system, and let $A_{ij} = A_i \parallel A_j$ be a pair-system of A, where $i, j \in \{1, \ldots, n\}, i \neq j$. Then, if $A_{ij} \models_\Phi f_{ij}$ for some LTL-X formula f_{ij} whose atomic propositions are all drawn from $prop(A_{ij})$, we would like to also conclude $A \models_\Phi f_{ij}$. For safety properties, this follows from execution projection. For liveness properties, we need something more, since the projection of an infinite execution of A onto A_{ij} could be a finite execution of A_{ij}, and so the liveness property in question may not be satisfied along this finite projection, while it is satisfied along all the infinite extensions. We therefore require that along an infinite global execution α, for every pair-system A_{ij}, an action involving A_{ij} occurs infinitely often along α. Write $A, \alpha \models \overset{\infty}{\mathsf{F}} ex(A_{ij})$ iff α contains infinitely many actions in which A_i or A_j or both participate in (this implies that α is infinite). Write $A, s \models_\Phi \overset{\infty}{\mathsf{F}} ex(A_{ij})$ iff for every infinite fair execution α starting in s: $A, \alpha \models \overset{\infty}{\mathsf{F}} ex(A_{ij})$. If $s = \langle s_1, \ldots, s_n \rangle$ is a state of A, then define $s \restriction ij = \langle s_i, s_j \rangle$, i.e., $s \restriction ij$ is the projection of s onto the pair-system A_{ij}.

Theorem 1. *Let $A = A_1 \parallel \cdots \parallel A_n$ be an I/O automaton. Let $i, j \in \{1, \ldots, n\}$, $i \neq j$, and let $A_{ij} = A_i \parallel A_j$. Assume that $A, u \models \overset{\infty}{\mathsf{F}} ex(A_{ij})$ for every start state u of A. Let s be a reachable state of A, and let f_{ij} be an LTL-X formula over $prop(A_{ij})$. Then*

$$A_{ij}, s \restriction ij \models_\Phi f_{ij} \;\; \textit{implies} \;\; A, s \models_\Phi f_{ij}.$$

The proof of Theorem 1 is available in the full version of the paper.[7] By applying the above result to all pair-programs, we obtain:

$$\bigwedge_{ij} (A_{ij} \models_\Phi f_{ij}) \text{ implies } A \models_\Phi \bigwedge_{ij} f_{ij}.$$

We then show that the conjunction $\bigwedge_{ij} f_{ij}$ of all the pair-properties implies the required global correctness property f, i.e., $(\bigwedge_{ij} f_{ij}) \Rightarrow f$. This leads to the following rule of inference:

$$\frac{\bigwedge_{ij} (A_{ij} \models_\Phi f_{ij}) \qquad (\bigwedge_{ij} f_{ij}) \Rightarrow f}{A \models_\Phi f}$$

4.1 Characterizing the global properties that can be verified

A natural question that arises is: how much verification power do we give up by the restriction to pairs? Are there interesting global properties that cannot be verified using our approach?

[7] `http://www.cs.virginia.edu/~lorenz/papers/cbse06/`

Compatibility:	Interface	Automaton	Behavioral
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 pair-wise behavior	complete but impractical

Table 1. The interoperability space for components

Let e_i ($i \geq 1$) denote an event, i.e., the execution of an action. Let $part(e_i)$ denote the components that participate in e_i. With respect to safety, we consider event ordering, i.e., $e_1 < e_n$, meaning that if e_1 and e_n both occur, then e_1 occurs before e_n. This can be verified by finding events $e_2, \ldots, e_{n_1}$ such that, for all $i = 1, \ldots, n-1$, $e_i < e_{i+1}$ can be verified in some pair. That is, there exist components $A \in part(e_i)$, $A' \in part(e_{i+1})$, and $A \parallel A'$ is a pair system that satisfies $e_i < e_{i+1}$. With respect to liveness, we consider leads-to properties, i.e., $e_1 \leadsto e_n$, meaning that if e_1 occurs, then e_n subsequently occurs. This can be verified by a similar strategy as outlined above for $e_1 < e_n$. Event ordering is sufficiently powerful to express many safety properties of interest, including mutual exclusion, FIFO, and priority. Leads-to is sufficiently powerful to express many liveness properties of interest, including absence of starvation and response to requests for service.

More generally, any global property that can be expressed by an LTL formula f which is deducible from pair-formulae f_{ij} can be verified. A topic of future work is to characterize this class of LTL formulae exactly.

4.2 Behavioral automaton of a component

A behavioral automaton 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" behavioral automaton for the component (which includes every transition the component makes, even internal ones), to a trivial automaton consisting of a single state. Thus, any behavioral automaton 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, to exporting just the interface (trivial automaton)—black box. Table 1 displays this spectrum.

The behavioral automaton 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 behavioral automaton, i.e., that it is truly a homomorphic image of the maximal automaton. Alternatively, the component compiler can generate a behavioral automaton from the code, using, for example, abstract interpretation or machine learning [32]. In this case, the behavioral automaton will be correct by construction. We assume the behavioral automaton for third party components is provided by the component designer.

4.3 Behavioral properties of a pair-program

In general, we are interested in behavioral properties that are expressed over many components at once. We infer such properties from the verified pair-properties. Such inference can be carried out, for example, in a suitable deductive system for temporal logic. The third-party assembler would have to specify the pair-properties and the pairs of interacting components and then carry out the deduction.

It is usually the case that the pairs of interacting processes are easily identifiable just based on the nature of process interactions in a distributed system. For example, in mutual exclusion, a pair-program is two arbitrary processes; in the elevator example the pair-program involves a floor component and an elevator controller component. Sometimes pair-properties to be verified are the same as the global specification, just projected onto a pair. For example, in mutual exclusion, the global property is just the quantification over all pairs of the pair-property given for some arbitrary processes i and j, i.e., $\bigwedge_{ij} \mathsf{G}\neg(C_i \wedge C_j)$, where C_i is a state of a process P_i corresponding to this process being in the critical section.

However, sometimes pair-properties are not straightforward projections of the required global properties. These pair-properties have to be derived manually. Then we have to prove that the conjunction of these pair-properties implies the global specification by the means of temporal logic deductive systems [17]. These proofs are usually quite small (e.g., 27 lines for the elevator example) and straightforward.

4.4 Verification of behavioral properties of the large program

At the verification stage the component assembler would have to choose a model-checker which he plans to do verification in, then provide to the model-checker a description of a behavioral automaton of the pair-program and the pair-properties in a suitable format. If verification is successful then the pair-properties hold in the global program and, as proven during the assembly phase, conjunction of these pair-properties implies the global property of the program. If verification is not successful then the third party assembler would have to either swap in a different component and repeat verification process or change the global property to be verified.

5 Implementation: Pair-wise Component Builder

We now describe the working of a pairwise verification methodology in a *pair-wise component builder* tool. This tool allows for interactive component design and pair-wise verification. The pair-wise builder is based on Sun's Bean Devel-

opment Kit (BDK) and the ContextBox [27, 26] software. Verification is done using the Spin model-checker [21] that uses LTL as a specification language and Promela as a modeling language. The goal is to provide the user with a list of design recommendations that should be considered when developing components in order to be able to use the tool for the subsequent component composition and verification.

A builder is used for creating a compound component out of subcomponents. The builder hence has two main functions:

- governing the connecting activity and dealing with problems of *interoperability*; and
- *encapsulating* the assembled components into the compound component.

Traditionally, builders focus on interoperability in the restricted sense of only considering interface compatibility, and support for system behavior prediction [14] is not available. In our framework, behavioral compatibility can also be checked. Hence, within our framework we implement a stronger notion of a builder, which, in addition to interface compatibility, can also deal with:

- *temporal behavior of connectors*, since we accommodate a stronger notion of interoperability, which encompasses both the interface between components and the temporal behavior of their connection (i.e., pair-properties), and
- *global temporal behavior*, that is, the temporal behavior of the assembled system. The deductive proofs that infer such properties of this global behavior from the known pair-properties are carried out within the builder.

For a component to be pair-wise composable in our builder, one takes the following steps. The component interface must be a collection of separate interfaces, each interface addressing a related connector. This corresponds to a JavaBeans component implementing several event listener interfaces, but does not need to be necessarily fine grained. This interface separation enables capturing only interface information relevant to a pair-system, which is model-checked during third-party assembly.

The inputs to our builder tool are components that have separate interfaces per connector. In the BDK this corresponds to a component's BeanInfo having a list of named operations that can be invoked by events. These functions are grouped and labeled according to components whose events this component subscribes to. Pair-wise composable components, as part of their BeanInfo, also have a high level description in Promela of their behavioral automata. The components are connected in the builder. As a result, relevant changes are made to their state machines to reflect components subscribing to each others events (i.e., Promela code of a pair-program is generated based on the interfaces and the behavioral automata of the pair).

Depending on component assembly, the user specifies properties of the model (as LTL formulae) that she wishes to verify. Properties can be over a single component or over a pair of components that were connected. The builder communicates the LTL specification and the generated Promela description of the

pair-program to the Spin interface. Then Spin model-checks the LTL formulae. Since model checking is over pair-systems, it is efficient. If violation of a property is detected, the user can modify either (1) the component, or (2) the component system design or (3) the properties, and then repeat the process.

6 Case Study: Elevator System

We now present a case study of a component-based algorithm that was pair-wise verified using our pair-wise component builder. This component-based elevator algorithm was implemented with a collection of interfaces to enable pair-wise verification. The elevator model consists of four types of components: *floor components* (numbered 1 to N), *panel button components*, the *user component*, and the *controller component*. The controller component (*controller*) represents the elevator, and hence implements the elevator's movement. Each floor component (*floor(f)*) represents a specific floor and maintains information about the floor and requests for this floor. Each panel button component (*panelbutton(f)*) represents a specific panel button inside an elevator. The events that floor components and controller component listen to (up requests and down requests for each floor) are generated by the user component (*user*) and by the buttons on the panel from inside the elevator.

6.1 Informal description

When moving upwards, the elevator controller satisfies requests in the upwards direction up to where it perceives the uppermost currently outstanding request to be, where this can be a panel request or a floor request in either direction. The elevator then reverses its direction and proceeds to satisfy the requests in the downwards direction until reaching what it perceives to be the lowermost currently outstanding request, then reverses again, etc. As the elevator passes floor f, it checks in with the *floor(f)* controller to determine if it needs to stop floor f due to an outstanding request in the direction of its movement, and stops if *floor(f)* so indicates. The *controller* maintains the following information.
Controller (elevator cabin):

`int g`—current elevator location
`int top`—the uppermost requested floor, set to 0 when this floor is reached
`int bottom`—the lowermost requested floor, set to 0 when this floor is reached
`boolean up`—true if the direction of the elevator movement is up, false if down
`boolean stop`—true if the elevator is not moving

Upon reaching the uppermost (lowermost) requested floor, the controller sets `top` (`bottom`) to 0, which is an invalid floor number. This indicates that the uppermost (lowermost) request currently known to the controller has been satisfied.

When a request for floor f is issued, an event is sent to the *floor(f)* component, where records the request, and also to the controller, which updates its `top` and `bottom` variables if necessary.

There are N floor components. Each floor component's state captures whether floor f is requested in either direction.
Floor(f), ($f = 1, \ldots, N$):

`bool up(f)`—true if the floor is requested for a stop in the upwards direction
`bool down(f)`—true if the floor is requested for a stop in the downwards direction

There are three types of event generators (button up at the floor, button down at the floor, floor number button on the panel), but only two event types (requests) are generated.

down button pushed at a floor$f$ generates a `downwards(f)` request.
up button pushed at a floor f generates an `upwards(f)` request.
floor f pushed on the panel inside an elevator, generates either a `upwards(f)` or `downwards(f)` request based on the elevator position.

The `upwards(f)` and `downwards(f)` events are randomly generated by a user component and N separate panel-button components that implement the panel buttons. The correctness of the algorithm obviously depends on correct maintenance of the `top` and `bottom` variables so that none of the requests are lost. It is important to update these variables not only when the new requests are issued but also to make sure they get reset after being reached, so that no subsequent requests are lost. Our update algorithm guarantees that all the requests are taken into account (only once).

6.2 Specification

There are two requirements that the elevator model must satisfy. (1) Safety: an elevator does not attempt to change direction without stopping first, since this would damage the motor, and (2) Liveness: every request for service is eventually satisfied

When connecting the controller component to the floor component, the builder would typically generate a CommandListener adapter, which subscribes to request events, so when an upwards request event is received from the floor, it invokes the up method of the controller. Now, the motor, due to physical constraints, cannot be switched from going up to going down without stopping first. Builders in current systems would not detect if the request sequence violated this constraint, and consequently the motor could be damaged at runtime. The safety property that we verified is the following:

If the elevator is moving up (down) then it continues doing so until it stops. The LTL formula for this is:

$$\mathsf{G}(\mathtt{up} \Rightarrow \mathtt{up}\mathsf{U_w}\mathtt{stop}) \ \wedge \ \mathsf{G}(\neg\mathtt{up} \Rightarrow (\neg\mathtt{up})\mathsf{U_w}\mathtt{stop}) \tag{1}$$

Where `boolean up` indicates direction of elevator movement and `boolean stop` indicates whether the elevator is moving or not. Recall that G is the "always" modality: $\mathsf{G}p$ means that p holds in all states from the current one onwards. $\mathsf{U_w}$ is "weak until": either p holds from now on, or q eventually holds, and p holds until then. This property can be verified by checking the controller alone, and so is not challenging, since the controller by itself has only a small number of states.

The interesting property that we verified is liveness: if a request is issued, it is eventually satisfied. The LTL formulae for this are:

$$\mathsf{G}(\mathtt{up}(\mathtt{f}) \Rightarrow \mathsf{F}(\mathtt{g} = \mathtt{f} \wedge \mathtt{stop} \wedge \mathtt{up})) \tag{2}$$

$$\mathsf{G}(\texttt{down(f)} \Rightarrow \mathsf{F}(\texttt{g} = \texttt{f} \wedge \texttt{stop} \wedge \neg\texttt{up})) \tag{3}$$

Where `g` is the elevator location variable and `f` is the number of the requested floor; `up(f)` and `down(f)` indicate request for floor f in a certain direction. $\mathsf{F}p$ means that p eventually holds.

6.3 Model-Checking

Our interconnection scheme enables the separation of the state spaces of the various components. Instead of constructing the global product automaton of N *floor components*, the *user component*, N *panel buttons*, and the *controller component* in the elevator system, we only construct N different *pair-machines*, and model-check each pair-machine separately. A *pair-machine* consists of a *controller component* and a single *floor component*, for each floor. We can then verify behavioral properties of each of these pair-machines in isolation, and then combine these properties deductively to obtain properties of the overall system.

Safety. Safety is a property local to the *controller component*, hence, it can be verified within the component. Since the *controller component* was model-checked in isolation, we needed to ensure that we are model-checking it in a correct environment (i.e., random request sequences). This is done through *input-enabledness* of the behavioral automaton. This ensures that if there is a transition that depends on the input from some other component we replace this transition by a set of transitions that embody all the possible input values from the other component (i.e., one transition per input value). The *controller component* non-deterministically chooses one of these transitions. During model-checking such input-enabledness creates an environment that produces the same execution traces as the product behavioral automaton of the two components. However, we avoid the extra states that would have been contributed by the second component that are unnecessary for this verification confined only to this component.

Liveness. We verified that liveness holds in our model by checking the model against the LTL formula (2). We verified liveness in the model with various number of floors N. This amounted to model-checking a system with N *floor(f)*, N *panelbutton(f)*, 1 *controller*, and 1 *user*.

To achieve pair-wise verification we needed to decompose the liveness property into pair-properties. Our pair consisted of a controller component and a floor component, where our pair-model was input-enabled for the input coming from other components. The pair-properties were manually derived for this pair-program. The global liveness property (the LTL formulae (2, 3)) of a system as a whole was deduced from the conjunction of the following pair-properties. These pair-properties were checked for each of N pairs *controller* $\|$ *floor(f)*. Define $p \rightsquigarrow q = \mathsf{G}(p \Rightarrow \mathsf{F}q)$.

When request is issued, it gets processed by the controller:
(p 1.1) $up(f) \rightsquigarrow top \geq f \bigwedge up(f) \bigwedge bottom \leq f$
(p 1.2) $down(f) \rightsquigarrow top \geq f \bigwedge down(f) \bigwedge bottom \leq f$

The *controller* actually moves up or down without changing its direction:

(p 2.1) $(g = g_0 < f \bigwedge up(f) \bigwedge up \bigwedge bottom \leq f \leq top) \rightsquigarrow$
$(g = g_0 + 1 \leq f \bigwedge up(f) \bigwedge up \bigwedge bottom \leq f \leq top)$

(p 2.2) $(g = g_0 \geq f \bigwedge up(f) \bigwedge \neg up \bigwedge bottom \leq f \leq top) \rightsquigarrow$
$(g = g_0 - 1 \leq f \bigwedge up(f) \bigwedge \neg up \bigwedge bottom \leq f \leq top)$

(p 2.3) $(g = g_0 < f \bigwedge down(f) \bigwedge up \bigwedge bottom \leq f \leq top) \rightsquigarrow$
$(g = g_0 + 1 \leq f \bigwedge down(f) \bigwedge up \bigwedge bottom \leq f \leq top)$

(p 2.4) $(g = g_0 \geq f \bigwedge down(f) \bigwedge \neg up \bigwedge bottom \leq f \leq top) \rightsquigarrow$
$(g = g_0 - 1 \leq f \bigwedge down(f) \bigwedge \neg up \bigwedge bottom \leq f \leq top)$

The *controller* stops once reaching the requested floor:

(p 3.1) $(up(f) \bigwedge up \bigwedge g = f) \rightsquigarrow (stop \bigwedge g = f \bigwedge \neg up(f))$

(p 3.2) $(down(f) \bigwedge \neg up \bigwedge g = f) \rightsquigarrow (stop \bigwedge g = f \bigwedge \neg down(f))$

The *controller* reverses direction at the top and the bottom:

(p 4.1) $(g = bottom \bigwedge up(f) \bigwedge \neg up \bigwedge bottom \leq f \leq top) \rightsquigarrow$
$(g = bottom \bigwedge up(f) \bigwedge up \bigwedge bottom \leq f \leq top)$

(p 4.2) $(g = top \bigwedge up(f) \bigwedge up \bigwedge bottom \leq f \leq top) \rightsquigarrow$
$(g = top \bigwedge up(f) \bigwedge \neg up \bigwedge bottom \leq f \leq top)$

(p 4.3) $(g = bottom \bigwedge down(f) \bigwedge \neg up \bigwedge bottom \leq f \leq top) \rightsquigarrow$
$(g = bottom \bigwedge down(f) \bigwedge up \bigwedge bottom \leq f \leq top)$

(p 4.4) $(g = top \bigwedge down(f) \bigwedge up \bigwedge bottom \leq f \leq top) \rightsquigarrow$
$(g = top \bigwedge down(f) \bigwedge \neg up \bigwedge bottom \leq f \leq top)$

Boundary (efficiency) condition

(p 5) $\mathsf{G}(bottom \leq g \leq top)$

6.4 Verification Results

We constructed models with $N = 3, 5, 7, 9, 10, 12, 30, 100$, where N is the number of floors. The systems were model-checked as a whole system and pair-wise.[8] The number of states in a whole system grew too large for Spin to model-check above $N = 10$. On the other hand, model-checking pair-wise was achieved up to $N = 100$ without experiencing exponential blow up. In the verification results (Table 2), "number of transitions" is the number of transitions explored in the search. This is indicative of the amount of work performed in model-checking the given properties, for the given value of N. Pair-wise model checking first suffers from some overhead due to the Spin model representation for $N \leq 10$, and then shows a polynomial increase in N, as expected from the theoretical analysis.

7 Conclusion and Related Work

Component-based systems are widely acknowledged as a promising approach to constructing large-scale complex software systems. A key requirement of a successful methodology for assembling such systems is to ensure the behavioral

[8] 1.7 GHz with 8Gb of RAM

Number of Floors (N)	Whole system		Pair-wise	
	# Transitions	Run Time	# Transitions	Run Time
3	3.6×10^8	45	4.6×10^3	1
5	5.6×10^8	74	5.1×10^4	1
7	7.6×10^8	121	2.4×10^5	1
9	9.4×10^8	169	7.7×10^5	1
10	8.9×10^8	170	1.2×10^6	1
12	N/A	N/A	2.9×10^6	1
30	N/A	N/A	1.9×10^8	8
100	N/A	N/A	1.9×10^9	109

Table 2. Spin model-checking result for an elevator system

compatibility of the components with each other. This paper presented a first step towards a *practical* method for achieving this.

We have presented a methodology for designing components so that they can be composed in a pair-wise manner, and their temporal behavior properties verified without state-explosion. Components are only required to have interface separation per connector to enable disentanglement of intercomponent communication and specification of the externally visible behavior of each component as a behavioral automaton.

Vanderperren and Wydaeghe [40, 35, 39, 36, 37] 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 [2] have defined a notion of interface automaton, and have developed a method for mechanically verify 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.

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 [5], the dining and drinking philosophers problems [4], an eventually serializable data service [6] and a fault-tolerant distributed shared memory [3]. Release of the pair-wise component builder will contribute to the ease of the exploitation of our methodology and its subsequent application.

References

[1] Abadi, M., Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and Systems **15**(1) (1993) 73–132

[2] de Alfaro, L., Henzinger, T.A.: Interface automata. (2001) In *Proceedings of the 9th Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM.

[3] Attie, P.C., Chockler, H.: Automatic verification of fault-tolerant register emulations. In: Proceedings of the Infinity 2005 workshop. (2005)
[4] Attie, P.C., Emerson, E.A.: Synthesis of concurrent systems with many similar processes. ACM Transactions on Programming Languages and Systems **20**(1) (1998) 51–115
[5] Attie, P.C.: Synthesis of large concurrent programs via pairwise composition. In: CONCUR'99: 10^{th} International Conference on Concurrency Theory. Number 1664 in LNCS, Springer-Verlag (1999)
[6] Attie, P.C.: Synthesis of large dynamic concurrent programs from dynamic specifications. Technical report, American University of Beirut (2005)
[7] Attie, P.C., Lorenz, D.H.: Correctness of model-based component composition without state explosion. In: ECOOP 2003 Workshop on Correctness of Model-based Software Composition. (2003)
[8] Cheung, S., Giannakopoulou, D., Kramer, J.: Verification of liveness properties in compositional reachability analysis. In: 5^{th} ACM SIGSOFT Symposium on the Foundations of Software Engineering / 6^{th} European Software Engineering Conference (FSE / ESEC '97), Zurich (1997)
[9] Cheung, S., Kramer, J.: Checking subsystem safety properties in compositional reachability analysis. In: Proceedings of the 18^{th} International Conference on Software Engineering, Berlin, Germany, ICSE 1996, IEEE Computer Society (1996)
[10] Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design **9**(2) (1996)
[11] Clarke, E.M., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. In: Proceedings of the 5^{th} International Conference on Computer Aided Verification. Volume 697 of LNCS., Berlin, Springer-Verlag (1993) 450–462
[12] Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems **16**(5) (1994) 1512–1542
[13] Clarke, E.M., Long, D., McMillan, K.L.: Compositional model checking. In: Proceedings of the 4^{th} IEEE Symposium on Logic in Computer Science, New York, IEEE (1989)
[14] Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K., eds.: Proceedings of the 4^{th} ICSE Workshop on Component-Based Software Engineering: Component Certifiation and System Prediction. In Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K., eds.: Proceedings of the 4^{th} ICSE Workshop on Component-Based Software Engineering: Component Certifiation and System Prediction, Toronto, Canada, ICSE 2001, IEEE Computer Society (2001)
[15] Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Proceedings of the 5^{th} International Conference on Computer Aided Verification. Volume 697 of LNCS., Berlin, Springer-Verlag (1993) 463–477
[16] Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design: An International Journal **9**(1/2) (1996) 105–131
[17] Emerson, E.A.: Temporal and modal logic. In Leeuwen, J.V., ed.: Handbook of Theoretical Computer Science. Volume B, *Formal Models and Semantics*. The MIT Press/Elsevier, Cambridge, Mass. (1990)
[18] Grumberg, O., Long, D.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems **16**(3) (1994) 843–871
[19] Heineman, G.T., Councill, W.T., eds.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley (2001)
[20] Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You assume, we guarantee: Methodology and case studies. In: Proceedings of the 10^{th} International Conference on Computer-Aided Verification (CAV). (1998)

[21] Holzmann, G.J.: The SPIN Model Checker. Addison Wesley, San Francisco, California, USA (2003)
[22] ICSE 2001: Proceedings of the 23^{rd} International Conference on Software Engineering, Toronto, Canada, ICSE 2001, IEEE Computer Society (2001)
[23] Kesten, Y., Pnueli, A.: Verification by augmented finitary abstraction. Information and Computation **163**(1) (2000) 203–243
[24] Kesten, Y., Pnueli, A., Vardi, M.Y.: Verification by augmented abstraction: The automata-theoretic view. Journal of Computer and System Sciences **62**(4) (2001) 668–690
[25] Lamport, L.: Composition: A way to make proofs harder. In: Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium), Zurich (1997)
[26] Lorenz, D.H., Petkovic, P.: ContextBox: A visual builder for context beans (extended abstract). In: Proceedings of the 15^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Minneapolis, Minnesota, OOPSLA'00, ACM SIGPLAN Notices (2000) 75–76
[27] Lorenz, D.H., Petkovic, P.: Design-time assembly of runtime containment components. In Li, Q., Firesmith, D., Riehle, R., Pour, G., Meyer, B., eds.: Proceedings of the 34^{th} International Conference on Technology of Object-Oriented Languages and Systems, Santa Barbara, CA, IEEE Computer Society (2000) 195–204
[28] Lorenz, D.H., Vlissides, J.: Designing components versus objects: A transformational approach. In: ICSE 2001 [22] 253–262
[29] Lynch, N., Tuttle, M.: An introduction to input/output automata. Technical Report CWI-Quarterly, 2(3):219–246, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands (1989)
[30] Lynch, N., Vaandrager, F.: Forward and backward simulations — part I: Untimed systems. Information and Computation **121**(2) (1995) 214–233
[31] Lynch, N.A.: Distributed Algorithms. Morgan-Kaufmann, San Francisco, California, USA (1996)
[32] Mäkinen, E., Systä, T.: MAS - an interactive synthesizer to support behavioral modeling in UML. In: ICSE 2001 [22] 15–24
[33] Pnueli, A.: The temporal logic of programs. In: IEEE Symposium on Foundations of Computer Science, IEEE Press (1977) 46–57
[34] Szyperski, C.: Component Software, Beyond Object-Oriented Programming. Addison-Wesley (1997)
[35] Vanderperren, W., Wydaeghe, B.: Towards a new component composition process. In Proceedings of ECBS 2001 (2001)
[36] Vanderperren, W., Wydaeghe, B.: Separating concerns in a high-level component-based context. In EasyComp Workshop at ETAPS 2002 (2002)
[37] Vanderperren, W.: A pattern based approach to separate tangled concerns in component based development. In Coady, Y., ed.: Proceedings of the 1^{st} AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, Enschede, The Netherlands (2002) 71–75
[38] Wallnau, K.C., Hissam, S., Seacord, R.: Building Systems from Commercial Components. Software Engineering. Addison-Wesley (2001)
[39] Wydaeghe, B., Vandeperren, W.: Visual component composition using composition patterns. In Proceedings of Tools 2001 (2001)
[40] Wydaeghe, B.: PACOSUITE: Component composition basedon composition patterns and usage scenarios. PhD Thesis (2001)