Establishing Behavioral Compatibility of Software Components 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 the temporal behavior properties of such systems. Our verification method is mostly automatic, with very little manual deduction required. Our verification method is not susceptible to the well-known *state explosion* problem, which has hitherto severely limited thepractical applicability of automatic verification methods. Our method specifies the externally visible behavior of each component C as several interface automaton, one for each of the other components which C interacts directly with. For each pair of directly interacting components, we compute the product of the interface automata. These "pair machines" can then be verified mechanically, since they are small. The verified "pair properties" can then be combined to deduce global properties. This combination is done deductively, and is quite simple, since the hard work of verifying the pair-properties has already been done. Our case study of an elevator example will substantiate this point.

Another contributuion of this paper is the use of several interface automata per component. This enables a clean separation between interfaces, so that the interactions of a component C with several others are cleanly separated, and can be inspected in isolation. 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. We illustrate our method by designing and verifying an elevator example, a standard example in the software engineering literature. To our knowledge, our method is the first approach to behavioral compatibility that does not suffer from state-explosion.

1. INTRODUCTION

Monolithic software systems are fragile and unreliable. In principle, *component-based programming* (CBP) [25] alleviates this inherent software problem. Software components may be produced by third-party providers applying cost-effective product line development [6, 7]. Third-party composition of software systems is less susceptible to the developer's programming skills and improves the productivity of system developers [31]. Finally, black-box reuse of third-party components increases the confidence in their reliability, which in turn reduces the system's overall fragility, at least so it seems.

Software components 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 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 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 will address issues of behavioral compatibility amongst components, and will enable reasoning about the global behavior (including temporal behavior, i.e., safety and liveness) of an assembly of components. Our technique is not restricted to small, unrealistic applications. We illustrate the model concretely by means of an example design for an elevator system, which can scale up in size (number of components and number of states) and still be modeled checked.

1.1 Component interoperability

Components are "units of independent production, acquisition, and deployment" [25]. In *component-based software engineering* (CBSE) [11], software development is decoupled from assembly and deployment. During component devel-

[*]Supported in part by the National Science Foundation (NSF) under Grant No. CCR-0204432, and by the Institute for Complex Scientific Software at Northeastern University.

opment the component's source code is compiled, typically by third party component providers, and "forgotten" thereafter. That is, a component may originate from a different author in binary format and without its source code.

Third party assembly (henceforth, *third party composition*) is the activity of connecting components. During assembly time, the application (or component) is assembled from other (compiled) components. The activity takes place between the compilation of the components and the compilation (or serialization) 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 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 [22] 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 OOP where the design is centralized, but is not practical in component-based designs [20]. Agreement before hand is possible only if third-party components providers are coordinated.

Extending Parnas's principles to 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 before hand. For third-party composition to work, components need to agree of how to agree rather then agree on the interface.

Indeed, CBP builder environments (e.g., BeanBox [14], JavaStudio [16], PowerJ [23], JBuilder [17], VisualAge [29], VisualCafé [30]), typically apply two mechanisms to overcome this difficulty and support third-party composition. First, to check for interface compatibility, builders use *introspection*. Introspection is means for discovering the component interface. Second, builders support *adaptability* by generating adapters to overcome differences in the interface. Adapters are means of fixing small mismatches when the interfaces are not syntactically identical.

1.3 Behavioral compatibility

The goal of works 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 [32, 28, 33, 27, 26] 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 is based on theoretical automata theory but fail to provide a practical foundation for software growth (because of sate explosion.)

All current mechanical methods for reasoning about behavior (finite state systems) that we are aware of rely on some form of exhaustive state-space search to generate all the possible behaviors of the program. These methods are thus susceptible to ***state explosion***: the number of global states of a concurrent program consisting of K processes, each with $O(N)$ local states, is in $O(N^K)$. Thus, all extant methods have time complexity exponential in the number of components in the system. Thus they are not suitable for analyzing and constructing large complex systems. We address the challenge of avoiding state explosion. In Section 3, we show an elevator example which, when scalled up to 200 floors, requires an upper bound of only 1166400 states (instead of 10^{180}), which is within in the reach of 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. Chief among behavioral properties are:

- Safety properties: "nothing bad happens" — for example, when an elevator is moving up, it does not attempt to move down without stopping first.
- 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 unambiguously documents what the system is supposed to achieve. Proposed solutions to the problem of the design and implementation of large software systems can be generally classified as informal (i.e., *Computer-Aided Software Engineering*) versus formal:

- *Computer-Aided Software Engineering* (CASE). These approaches use a structured notation for describing large systems and tools to support editing and simulation. While CASE notations provide an operational semantics, and are certainly an improvement over current ad hoc methods, the refinement steps in CASE-based methodologies are typically informal in nature.

There is no assurance, apart from informal reasoning, that the refined system satisfies the same correctness properties as the original system.

- *Formal methods.* This approach addresses the main deficiency of CASE. The process of constructing software involves a formal proof of correctness. We classify formal methods into:

    1. *Proof-theoretic.* In proof-theoretic methods, a suitable deductive system is used, and correctness proofs are built manually, or using a theorem prover.
    2. *Model-theoretic.* In model-theoretic methods, a model of the run-time behavior of the software is built, and this model is checked (usually mechanically) for the required properties.

    Since the cost of the manual labor required is still one of the obstacles to wide-scale deployment of formal methods, we shall emphasize model-theoretic methods, due to their greater potential for automation.

    We can also classify formal methods into *analytic* and *synthetic* methods.

    1. *Analytic.* An analytic method takes an already constructed system and determines whether it satisfies a given specification. The most successful exemplar of this to date is *temporal logic model checking* [4].
    2. *Synthetic.* A synthetic method starts from the specification and derives a correct system. *Stepwise refinement* methodologies, where an implementation is derived by a series of refinement steps, are examples of this. Another example is *temporal logic synthesis methods*, where a correct (finite-state) program is mechanically produced from a specification expressed as a formula of some propositional temporal logic.

2.1 Avoiding state-explosion by pairwise composition

In [3, 1], we present a temporal logic synthesis method for the synthesis of finite-state concurrent programs from specifications expressed in the branching-time propositional temporal logic CTL [9]. The method of [3, 2] 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. The final program is generated by a syntactic composition of all the pair-programs. This composition has a conjunctive nature: a process P_i can make a transition if and only if that transition is permitted by ***every*** pair-machine in which P_i participates.

The pairwise method as given in [3, 2] 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 pairwise 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.

2.2 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 is accompanied by a *behavioral automaton* [13, 5], which 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 behavioral automata, we construct the pair-machine for their interaction by simply taking the automata-theoretic product of the behavioral automata (possibly first projecting these automata on the common inputs/outputs of the two components). 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.

2.3 The interoperability space for components

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 components 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. The proposed project falls in the middle column.

In practice, it is unrealistic to expect the programmer to provide the maximal behavioral automaton, 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.

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. Alter-

	Interface compatibility	Automaton 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 pair-wise behavior	complete but impractical

Table 1: The interoperability space for components

natively, the component compiler can generate a behavioral automaton from the code, using, for example, abstract interpretation or machine learning [21]. In this case, the behavioral automaton will be correct by construction. In this project, we will assume the first option for third party components, and will explore the second option for components assembled in our builder.

3. EXAMPLE

One of the traditional and most widely studied examples in software engineering is the "lift example": a building contains F floors and E elevators. Requests for service come from "panel" buttons inside an elevator, indicating a request from a passenger within the elevator, and from "floor" buttons on a particular floor, indicating a request a request from a passenger waiting at a particular floor. The latter requests have a direction (up or down) associated with them, and can be satisfied by any of the elevators.

In a system with several elevators operating concurrently, and with requests from the various floors arriving concurrently, the coordination of the elevators and the efficient resolution of the floor requests is quite intricate. Furthermore, a straightforward solution suffers from significant state-explosion, since the global state-space is the product for the state-spaces of all the elevators, (including all the panel requests), and all the floor requests.

3.1 Requirements

The problem is to design software for the various components of an elevator system. The requirements have both static aspects and dynamic aspects.

The static aspects are expressed as interfaces. The components implement the `CommandListener` interface:

```
public interface CommandListener extends EventListener {
  public void onCommand(CommandEvent e);
}
```

and the `RequestListener` interface:

```
public interface RequestListener extends EventListener {
  public void onRequest(RequestEvent e);
}
```

`CommandEvent` and `RequestEvent` are event classes. The payload in the `CommandEvent` specifies if the request is up or down or stop. The payload in the `RequestEvent` specifies the floor, and optionally up or down.

The dynamic aspects are given by the following required temporal behavioral properties: (1) Safety: an elevator does not attempt to change direction without stopping first, (2) Liveness: every request for service is eventually satisfied, and (3) Efficiency: for a request from a floor button, only one elevator responds.

3.2 Approaches and Architectures

We first consider current approaches to developing component-based systems. First, there are approaches based on a naive interface compatibility component implementation. Such "static approaches" merely ensure that the interfaces of two interacting components are compatible in that all method calls have the correct number and type of arguments, and the event types conform. While such approaches provide necessary checks, they are nevertheless not sufficient. They are akin to a compiler, which can ensure that a program is free of type errors, but cannot ensure that a program is free of bugs. Likewise, approaches that check only the static aspects of interfaces cannot ensure that the system has the correct behavioral properties. For example, checking that an elevator controller and an elevator motor have the correct interface with each other does not ensure that the controller will not send the motor a command to change direction without stopping first.

An improvement on static approaches are "dynamic approaches" which consider behavior compatability. Such methods typically express the externally visible temporal behavior of a component by means of a behavioral automaton. Then, the required temporal behavioral properties of the system as a whole are checked by computing the automata-theoretic product of behavioral automata of all the components, and then verifying the properties by means of exhaustive state-space search techniques, such as model-checking [4]. While the ability to verify temporal behavior is a major advance over the purely static methods, such dynamic approaches are severely limited in practice by the well-known *state-explosion* problem: if there are K behavioral automata, each with $O(N)$ states, then their product has $O(N^K)$ states. This exponential growth of the state space with the number K of components renders methods based on exhaustive state-space search inapplicable to any but the smallest systems, and hence of limited practical utility: such methods simply do not scale up.

To overcome state-explosion, we eschew the computation of the product of all K behavioral automata. Instead, we compute the products of *pairs* of behavioral automata, corresponding to the pairs of components that interact directly. In the worst case, where all components interact, this has complexity $O(N^2K^2)$. This low polynomial complexity means that our method scales up to large systems. We verify temporal behavior 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. Such combination can be by means of temporal logic deductive systems [9]. Since the pair-properties embody the complexity of the component interaction, this deductive part of the verification will be quite short. For example, in [1], a two-phase commit protocol is verified in a pairwise manner. The deductive argument to combine the pairwise properties into the standard correctness properties of two-phase commit consists of no more than 15 lines of proof.

3.3 Our Approach to Design of a Component-based System

The main insight is that components can be designed to enable pairwise composition, thus supporting behavioral computability checks that scale up to large complex systems. In terms of design of components this mean that the components are designed in such a way that we can check the system one connector at a time, i.e., pairwise. In this section we demonstrate this for a pairwise elevator component-system.

In order for a component to be pairwise composable, one has to take the following steps. The component interface needs to be a collection of separate interfaces, each interface addressing a related set of connectors. This corresponds to JavaBeans component implementing several event listener interfaces, but does not need to be necessarily fine grained. For the elevator example, we would expect the controller to have 3 interfaces: one for the communication channel with the elevator’s panel, one for the floor buttons, and for communication channel with the motor. We name those connector interfaces: panel interface, button interface, and motor interface.

Each connector interface is split into an external part and an internal part. The external interface plays a part in checking the behavioral compatibility with external components; the internal interface plays a part in checking the compatibility with the other internal ainterfaces. Both parts of the connector interface are designed such that they can be easily composed by superimposing the state transition graphs.

To reduce the complexity of varifying the composition, the aim during the design phase is to determin exactly what information needs to be captured in the local state of the component.

3.4 A Component-based Approach

When connecting the controller to the motor, the builder would typically generate `CommandListener` adapter, which subscribes to `up` events, and when an event is received from the controller, invokes the up method of the motor. Now, suppose the motor, for physical constraints, cannot be switched from going up to going down without stopping first. Builders in current systems would not detect if the controller violates this constraint, and consequently the motor could be “damaged” at runtime.

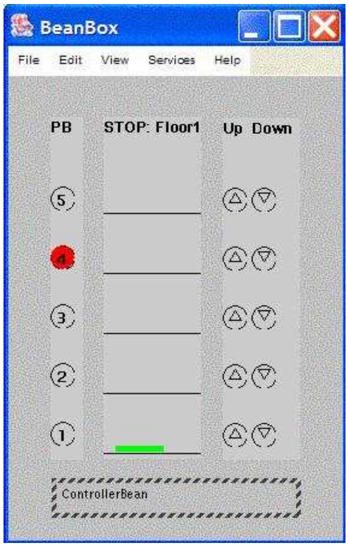


Figure 1: Elevator composed out of JavaBeans

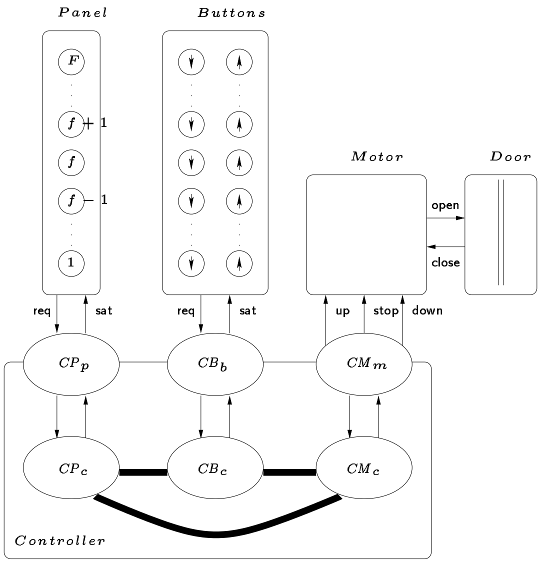


Figure 2: Elevator

In our framework, however, the motor component and the controller component would export their behavioral automata. The builder would compute the product of the two automata, and detect violation by model checking the pair product against the motor constraint which could be expressed as a temporal logic formula.

3.5 Implementation

The system is built from the JavaBeans components shown in Figure 1 (BDK does not render the connectors). Figure 2 gives the schematic architecture of our system. Our system consists of the following components among others:

- *Floor buttons:* For requesting the lift from outside. An up-button on floor f issues a request by the method call $\mathsf{uf\_req}(f)$, and receives notification that the request is satisfied by the method call $\mathsf{uf\_sat}(f)$. Likewise for the down button on floor f, with the method calls $\mathsf{df\_req}(f)$ and $\mathsf{df\_sat}(f)$, respectively.

- *Lift panel buttons:* For requesting, from inside, that the lift stop at a particular floor. A panel button for floor f issues a request by the method call $\mathsf{p\_req}(f)$, and receives notification that the request is satisfied by the method call $\mathsf{p\_sat}(f)$.

- *Lift controller:* Reads requests from panel and floor buttons. Send appropriate signals to the lift's motor and door. Has to coordinate with other lifts to assign floor requests.

- *Lift motor:* The motor sends the lift up and down, and stops the lift on designated floors. The motor responds to the method calls stop, up, down from the lift controller, which cause it to stop, start moving in the up direction, and start moving in the down directions, respectively.

- *Lift doors:* For opening and closing the entrance.

It is clear that the main challenge lies in designing the controller, since the buttons simply issue requests and receive responses, and the motor simply responds to motion commands. Figure 13 presents a centralized implementation of the controller. However, as discussed above, a centralized implementation is subject to state-explosion, since its state-space is the product of the state spaces of all components. Instead, we apply the pairwise method. We provide a separate interface between the controller and panel buttons (CP), the controller and floor buttons (CB), and the controller and elevator motor (CM). The interaction between the controller and each of these other components is modeled by a separate pair-machine. For sake of simplicity, our design will deal with only a single elevator. We subsequently outline how it can be expanded for multiple elevators. Figure 14 presents these separate interfaces.

Each interface is composed of two parts, which are composed together in a conjunctive manner (see Section 2.1). So, the controller-panel interface (on the controller side) CP consists of two automata CP_p and CP_c, given in Figures 3 and 4, respectively. We present automata as a directed graph, where the nodes are *local control states*, and the arcs are labeled with guarded commands [8], of the form $B \to A$, consisting of a guard B and an action A. If no guard is present, then the guard is taken to be *true*, and if no action is present, then the action is taken to be *skip*, i.e., a "no operation". CP_p interacts with the panel-buttons component, P, while CP_c interacts with the rest of the controller. Likewise, CB_b interacts with the floor-buttons component, B, while CB_c interacts with the rest of the controller, and CM_m interacts with the motor component, M, while CM_c interacts with the rest of the controller. Summarizing, we have the pair machines $CP_p \parallel_m P$, $CB_b \parallel_m B$, and $CM_m \parallel_m M$. The communication and coordination between the two processes in each pair-machine is by means of method calls, indicated by the m subscript in $\parallel_m$. In addition, the controller itself consists of three processes composed in parallel: $CP_c \parallel_i CB_c \parallel_i CM_c$. However, because these three processes are all part of the same component, their concurrent composition is "internal", and so a much "tighter" mechanism than method calls can be used. For example, CP_c, CB_c, CM_c can be "interpreted" by a central daemon, which generates the behavior of the controller. To retain maximum flexibility, we make no restriction in principle on the method of composition used within a component, and indicated by $\parallel_i$, the i standing for "internal." For the elevator example, we happen to use a shared memory model of concurrency. To express the "conjunctive" requirement of the pairwise method, we compose CP_p and CP_c in a conjunctive manner: CP_p and CP_c have the same local state structure, and the same arcs. They are required to execute arcs synchronously, i.e., an arc can be executed if and only if it is executed by both CP_p and CP_c. Likewise for CB_b/CB_c, and CM_m/CM_c.

We now provide a more detailed discussion of the example. We first discuss the interfaces CP_p, CB_b, and CM_m. We start with the panel interface CP_p.

Most local control states of CP_p are specified by four variables: (1) $floor \in 1 \dots F$, the current floor, (2) $dir \in \{u, d\}$, the direction of motion, u (up) or d (down), (3) $status \in \{s, m\}$, the status of motion, s (stopped) or m (moving), and (4) $requests \in \{p, i\}$, the existence of further requests in the direction of motion of the elevator, p indicates that such requests are pending, and i (idle) indicated no such requests are pending in the direction of motion, but some requests in the opposite direction are pending. By a request in the direction of motion of the elevator, we mean either that the elevator is at a floor f and moving up, and there is a request for a higher floor, or the elevator is at a floor f and moving down, and there is a request for a lower floor. In addition, there are states marked $[f, s, ii]$. This indicates being stopped at floor f, and with no requests *at all*, in either direction. These states are needed so that the elevator can stop moving when there are no requests pending anywhere.

The state transitions of CP_p are intended to accurately reflect the receipt of various requests from the panel. In addition to the local control states, CP_p maintains some "data" state, namely a set *panel_sched* of all of the panel requests received. *panel_sched* is updated by the method call $\mathsf{p\_req}(f)$, which adds f to *panel_sched*, and is also updated when the elevator stops at floor f: f is removed from *panel_sched*.

Figure 3 gives a "slice" of CP_p, corresponding to a single floor f, which is neither the top nor the bottom floor. The guards on the transitions are of two kinds. First, on every state, there are F "self-loop" transitions labeled with the guarded command $\mathsf{p\_req}(f) \to panel\_sched := panel\_sched \cup \{f\}$. These serve to update *panel_sched* upon receipt of a $\mathsf{p\_req}(f)$ method call. Second, there are transitions between states, whose guards are predicates over *panel_sched*. To express these succinctly, we define the following predicates: $r(f) = f \in panel\_sched$, i.e., a request at floor f, $a(f) = \exists f' : f' > f \wedge f' \in panel\_sched$, i.e., a request above floor f, $b(f) = \exists f' : f' < f \wedge f' \in panel\_sched$, i.e., a request below floor f. Transitions without a label are defined to have the label *true*.

We now explain briefly the transitions leaving each state in Figure 3. For some states, we use a sublist whose item headers are the guards of the transitions leaving that state. To save space, discuss only the transitions concerned with upward motion, and with reversing direction from up to down. The transitions for downward motion, and for reversing direction from down to up, are symmetric.

- $[f, u, m, p]$:
  - $\neg r(f) \wedge a(f)$, i.e., no request for the current floor but a request for a higher floor, then the elevator keeps moving up, so transition to $[f+1, u, m, p]$.
  - $r(f) \wedge a(f)$, i.e., a request for both the current floor and some higher floor, then the elevator first stops, so transition to $[f, u, s, p]$, and then continues on up, so transition from $[f, u, s, p]$ to $[f + 1, u, m, p]$.
  - $r(f) \wedge \neg a(f) \wedge b(f)$: i.e., a request for the current floor, no request for a higher floor, and a request for some lower floor. The elevator first stops, so transition to $[f, u, s, i]$, and then may either continue up, or may reverse direction. This will depend on what floor button requests are pending. We discuss this further below.
- $[f, u, s, p]$: Since more requests in the up direction are pending, the elevator continues moving up, so transition unconditionally (i.e., with guard *true*) to $[f + 1, u, m, p]$.
- $[f, u, m, i]$:
  - $\neg a(f)$, i.e., no request for the current floor but a request for a higher floor, then the elevator keeps moving up, so transition to $[f + 1, u, m, p]$.
  - $a(f)$, i.e., a request for both the current floor and some higher floor, then the elevator first stops, so transition to $[f, u, s, p]$, and then continues on up, so transition from $[f, u, s, p]$ to $[f + 1, u, m, p]$.
  - *true*, there is a transition with *true* guard from $[f, u, , i]$ to $[f, d, s, p]$ (reverse direction). Since, by virtue of $pending = i$ in $[f, u, s, i]$, there must be pending requests below, the latter transition is to a state with $pending = p$, i.e., $[f, d, s, p]$.
- $[f, u, s, i]$: Since there are no pending panel requests for floors above the current floor, the elevator can either continue moving up (in case there are floor button requests higher up), or can reverse direction. So, there are transitions with *true* guards from $[f, u, s, i]$ to $[f + 1, u, m, i]$ (continue moving up), and to $[f, d, s, p]$ (reverse direction). Since, by virtue of $pending = i$ in $[f, u, s, i]$, there must be pending requests below, the latter transition is to a state with $pending = p$, i.e., $[f, d, s, p]$.
- $[f, s, ii]$ In this state, there are no pending panel requests at all. So, receipt of a request for a higher floor, causing $a(f)$ to become true, enables the transition to $[f, u, s, p]$, and receipt of a request for a lower floor, causing $b(f)$ to become true, enables the transition to $[f, d, s, p]$.

The floor button interface CB_b operates in a similar manner to CP_p, except that the direction of the requests (request to go up, or request to go down) must be considered. Thus, we maintain two sets, *up_floor_sched*—the set of all floor requests in the up direction, and *down_floor_sched*—the set of all floor requests in the down direction. These are updated by the method calls $\mathsf{uf\_req}(f)$, $\mathsf{df\_req}(f)$, respectively, which insert a request for floor f. Floor f is removed from *up_floor_sched*, *down_floor_sched* when the elevator visits floor f and is traveling in the appropriate direction. To express the needed transition guards in CB_b succinctly, we define the following predicates: $ur(f) = f \in up\_floor\_sched$, i.e., an up request at floor f, $dr(f) = f \in down\_floor\_sched$, i.e., a down request at floor f, $a(f) = \exists f' : (f' \in up\_floor\_sched \vee f' \in down\_floor\_sched) \wedge f' > f$, i.e., a request above floor f, $b(f) = \exists f' : (f' \in up\_floor\_sched \vee f' \in down\_floor\_sched) \wedge f' < f$, i.e., a request below floor f. Transitions without a label are defined to have the label *true*. CB_b has the same local state structure as CP_p, and the same transition guards, except for the transitions that involve stopping at a floor. Here, the direction of requests is important. So, for example, the transition from $[f, u, m, p]$ to $[f, u, s, p]$ has guard $ur(f)$, rather than just $r(f)$, since the elevator should only stop for a floor up request when it is going up. Also, the method calls are adjusted appropriately, so the $[f, u, m, p]$ to $[f, u, s, p]$ transition issues the method call $\mathsf{uf\_sat}(f)$, for example.

The motor interface CM_m issues stop, up, and down method calls to the motor, which command it to stop the elevator at the current floor, start the elevator moving in the up directions, and start the elevator moving in the down direction, respectively. The states of CM_m are specified by the current floor f, the the status of motion, s or m, and the direction of motion u or d. CM_m issues method calls as appropriate to the transition being made, e.g., a transition from $[f, s, u]$ to $[f + 1, m, u]$ issues the method call up, since the elevator must move up in this case.

The three interfaces CP_p, CB_b, and CM_m are coordinated by the remaining parts of the controller, as follows. Each of the automata CP_p, CB_b, and CM_m is composed "conjunctively" with an "internal controller" counterpart CP_c, CB_c, and CM_c, respectively. Conjunctive composition means that the two automata has the same local state structure and the same transition structure, but can have different guarded commands labeling corresponding transitions. For example, the $[f, u, m, p]$ to $[f, u, s, p]$ transition in CP_p is labeled with the guarded command $r(f) \wedge a(f) \rightarrow \mathsf{p\_sat}(f)$, whilst the $[f, u, m, p]$ to $[f, u, s, p]$ transition in CP_c is labeled with the guarded command $CM.f$, which is true when the automaton CM_c is in a state which has a floor component equal to f. Then, both CP_p and CP_c must execute the transition from $[f, u, m, p]$ to $[f, u, s, p]$ simultaneously, i.e., both the guards $r(f) \wedge a(f)$ and $CM.f$ must hold, and execution must result in both guarded command bodies being executed (in this case the body of the second guarded command is null). Finally, to achieve the required coordination amongst the interfaces, the three automata CP_c, CB_c, and CM_c are tightly coupled by means of a shared memory communication and synchronization mechanism, whereby each automaton can inspect the local state of the other, and use this local state in evaluating guards on its transitions, e.g., as in the guard

$CM.f$ on the $[f, u, m, p]$ to $[f, u, s, p]$ transition in CP_c discussed above.

The general interconnection scheme is then as follows.

- CP_p and CP_c are composed "conjunctively," so that they take the same transitions simultaneously. One can think of the two automata as being "overlaid" on top of each other, and, if a transition in CP_p is labeled with guarded command $B_p \to A_p$, and the same transition in CP_c is labeled with $B_c \to A_c$, then the transition in the overlaid automaton is labeled with $B_p \wedge B_c \to A_p \parallel A_c$. See [3, 1] for a formal definition and extended discussion of this conjunctive composition. We let CP denote the result of this conjunctive composition.
- In a similar manner, CB_b and CB_c are composed conjunctively, resulting in CB, and CM_m and CM_c are composed conjunctively, resulting in CM.
- CP_p interacts with the panel P via the method calls p_req(f) and p_sat(f). CB_b interacts with the floor buttons B via the method calls uf_req(f), uf_sat(f), df_req(f), and df_sat(f). CM_m interacts with the motor M via the method calls stop, up, and down. These interactions are given by the pair-machines $CP_p \parallel_m P$, $CB_b \parallel_m B$, and $CM_m \parallel_m M$.
- CP_p, CB_b, and CM_m can read each others local state and condition their transitions on the local state of each other. This interaction is given by the composition $CP_c \parallel_i CB_c \parallel_i CM_c$.

Figure 2 presents this interconnection scheme. The heavy dark lines indicate the shared memory mechanism. Arrows labeled with method calls indicate interfaces. It remains to discuss the operation of $CP_c \parallel_i CB_c \parallel_i CM_c$.

We regard the motor controller CM as defining the physical location of the elevator, since it actually controls the motor. We regard CP and CB as defining "virtual" elevators, since each of them has a local control-state space that defines a current floor f, direction of motion (up or down), status of motion (stopped or moving), and further requests (more requests in the direction of motion, or not). In particular, the discussion of CP_p given above should be viewed with this in mind.

CP and CB are synchronized with CM via $CP_c \parallel_i CB_c \parallel_i CM_c$. This synchronization works as follows. All of CM, CP, and CB are never more than one floor apart. Except when a "crossover" (i.e., a change of direction) is in progress, all three machines are moving in the same direction. We discuss the up direction and crossover from up to down. The other cases are symmetric. Upon reaching a floor f whilst moving up, CM waits for CP and CB to make the next move. If both move on to floor $f+1$, then CM does to. If CP stops at floor f, then CM stops at floor f, since this indicates a panel request for floor f. Likewise, if CB stops at floor f, then CM stops at floor f, since this indicates a floor button request for floor f. If both CP and CB stop at floor f then again CM stops at f. After stopping at f, CM must decide whether to continue going up or to change direction. If the *requests* bit of either CP or CB is set to p, this indicates more panel or floor requests in the up direction (i.e., above the current floor f), and so CM continues moving up. if both *requests* bits are set to i, then this indicates no more requests in the up direction, but some requests in the down direction, and so CM changes direction. See Figure 8, and in particular, the guards on transitions leaving state $[f, s, u]$. if one of CP or CB has its *requests* bit set to i whilst the other is in the state $[f, s, ii]$ (indicating no requests at all), then again CM changes direction. If both CP and CB are in the state $[f, s, ii]$, then CM remains stationary at floor f, since in this case there are no pending requests in the system. When CP has its *requests* bit set to i, then it keeps moving up along with CM and CB, as needed, without stopping at any floors. Similar remarks apply to CB. This design allows CP, CB, and CM to remain always within one floor of each other, which greatly simplifies the design of $CP_c \parallel_i CB_c \parallel_i CM_c$, and also keeps the number of states of $CP_c \parallel_i CB_c \parallel_i CM_c$ in $O(F)$, rather than $O(F^3)$. The constraint of staying within one floor of each other is enforced by appropriate guards in the transitions of each automaton. When referencing the local state of one automaton within another, we use the notation "automaton-name.predicate". For example, the guard $CM.(f, u)$ which appears in Figure 4, is true iff $CM.floor = f \wedge CM.dir = u$, where $CM.floor$, $CM.dir$ are the $floor$ and dir variables in the local state space of CM. Similar abbreviation is used for all the other guards in CP_c, CB_c, and CM_c. We also use underscore to indicate "don't care." When we use less than 4 argument, it should be understood to mean that the omitted arguments are don't cares.

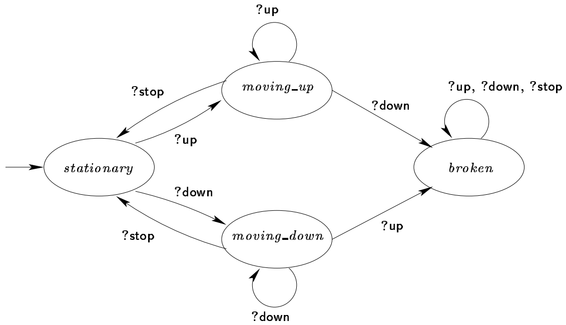


Figure 9: Motor

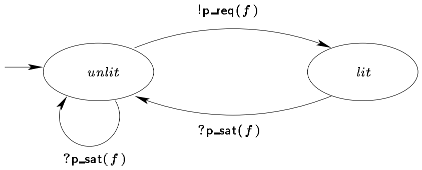


Figure 10: Panel Buttons

We can regard CP and CB as moving along one of eight "chains" of local states, given by the eight possible settings of the *dir*, *status*, and *requests* bits. A ninth chain consists of the states $[f, s, ii]$, indicating no requests at all, and in this case CP, CB just keep moving in concert with CM. These transitions out of $[f, s, ii]$ are not shown for sake of clarity of the diagrams.

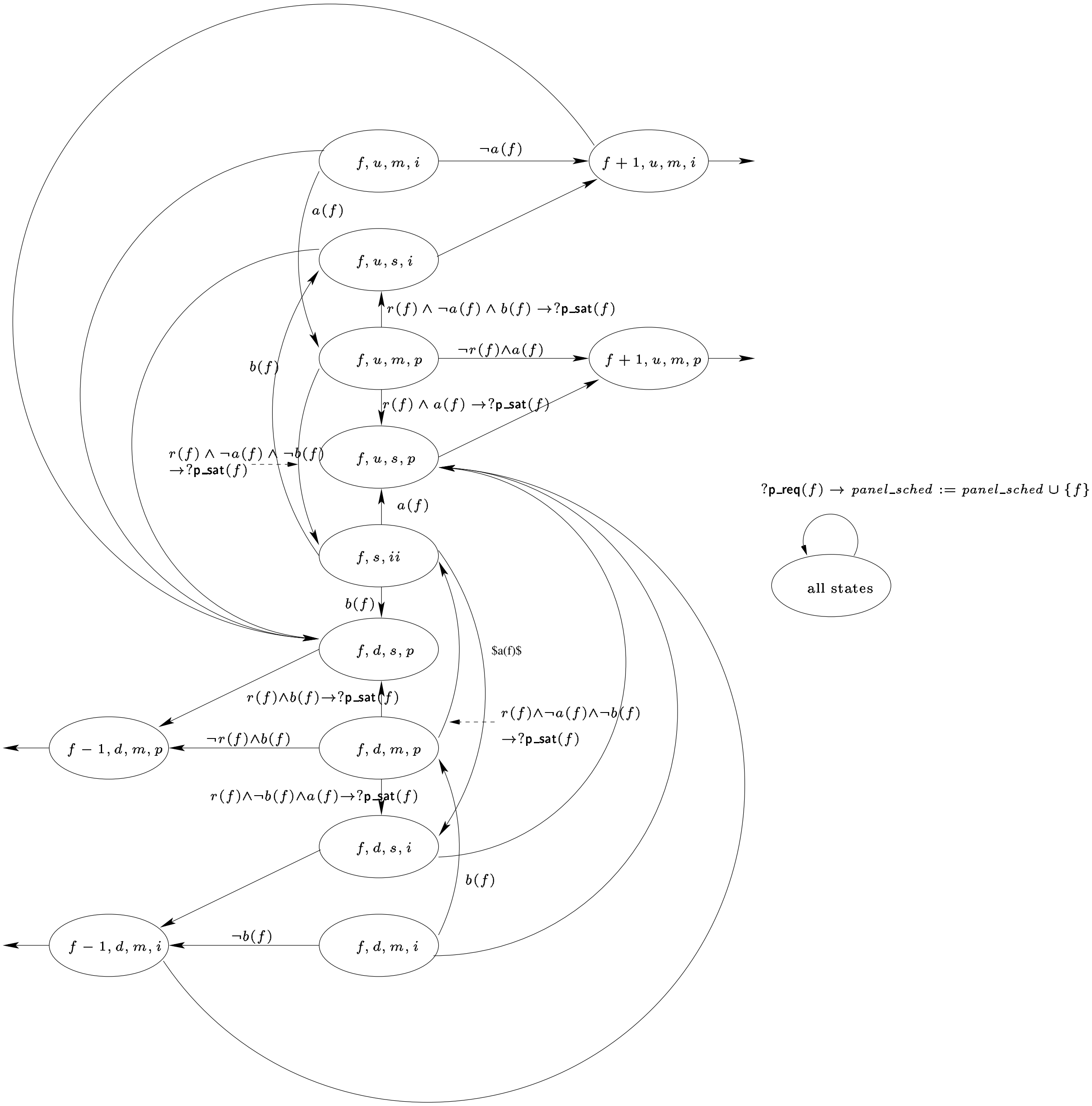

f, u, m, i
$\neg a(f)$
$f+1, u, m, i$
$a(f)$
f, u, s, i
$r(f) \wedge \neg a(f) \wedge b(f) \rightarrow ?\mathsf{p\_sat}(f)$
$b(f)$
f, u, m, p
$\neg r(f) \wedge a(f)$
$f+1, u, m, p$
$r(f) \wedge a(f) \rightarrow ?\mathsf{p\_sat}(f)$
$r(f) \wedge \neg a(f) \wedge \neg b(f)$
$\rightarrow ?\mathsf{p\_sat}(f)$
f, u, s, p
$?\mathsf{p\_req}(f) \rightarrow panel\_sched := panel\_sched \cup \{f\}$
$a(f)$
f, s, ii
all states
$b(f)$
f, d, s, p
$a(f)$
$r(f) \wedge b(f) \rightarrow ?\mathsf{p\_sat}(f)$
$r(f) \wedge \neg a(f) \wedge \neg b(f)$
$\rightarrow ?\mathsf{p\_sat}(f)$
$f-1, d, m, p$
$\neg r(f) \wedge b(f)$
f, d, m, p
$r(f) \wedge \neg b(f) \wedge a(f) \rightarrow ?\mathsf{p\_sat}(f)$
f, d, s, i
$b(f)$
$f-1, d, m, i$
$\neg b(f)$
f, d, m, i

Figure 3: CP_p

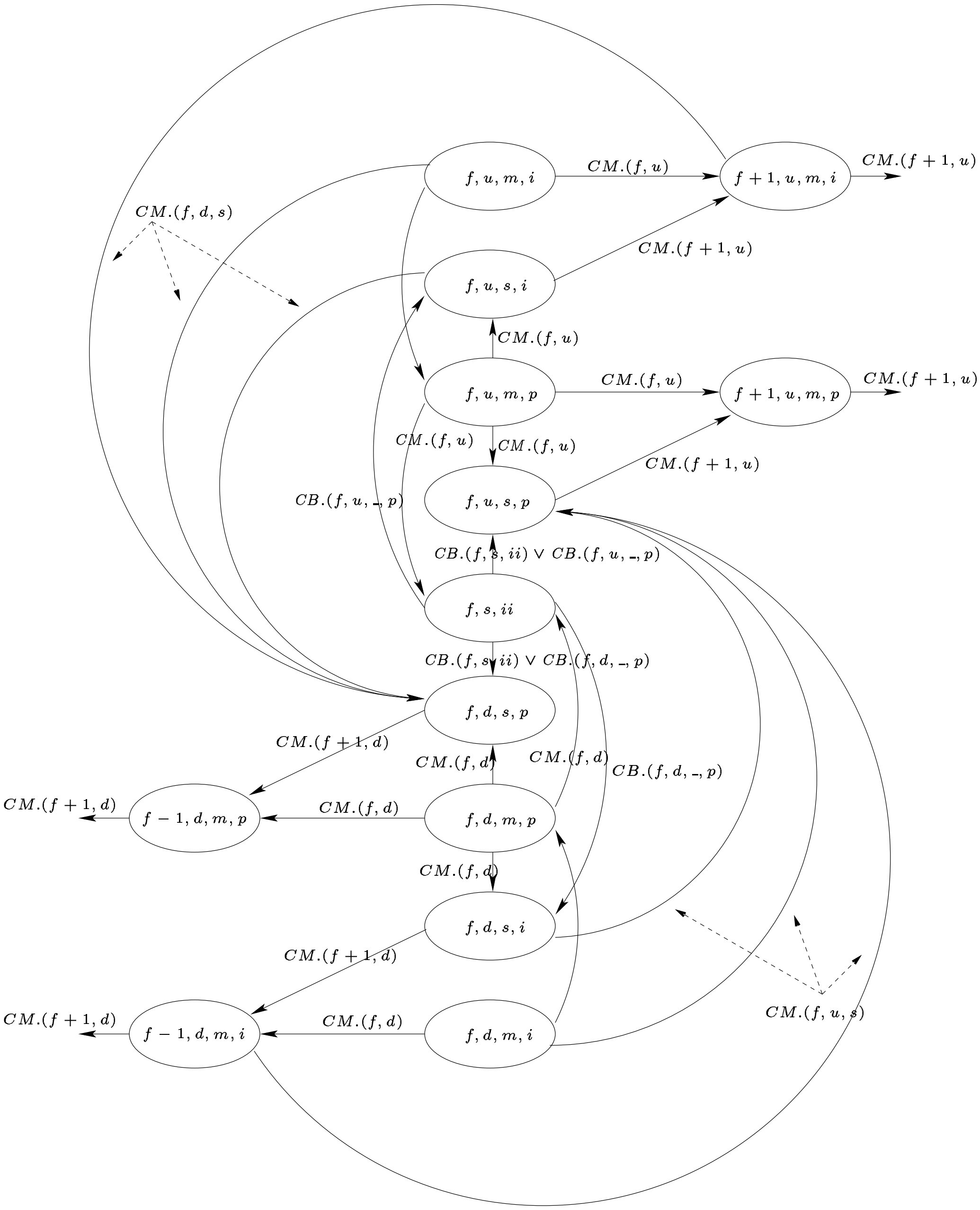
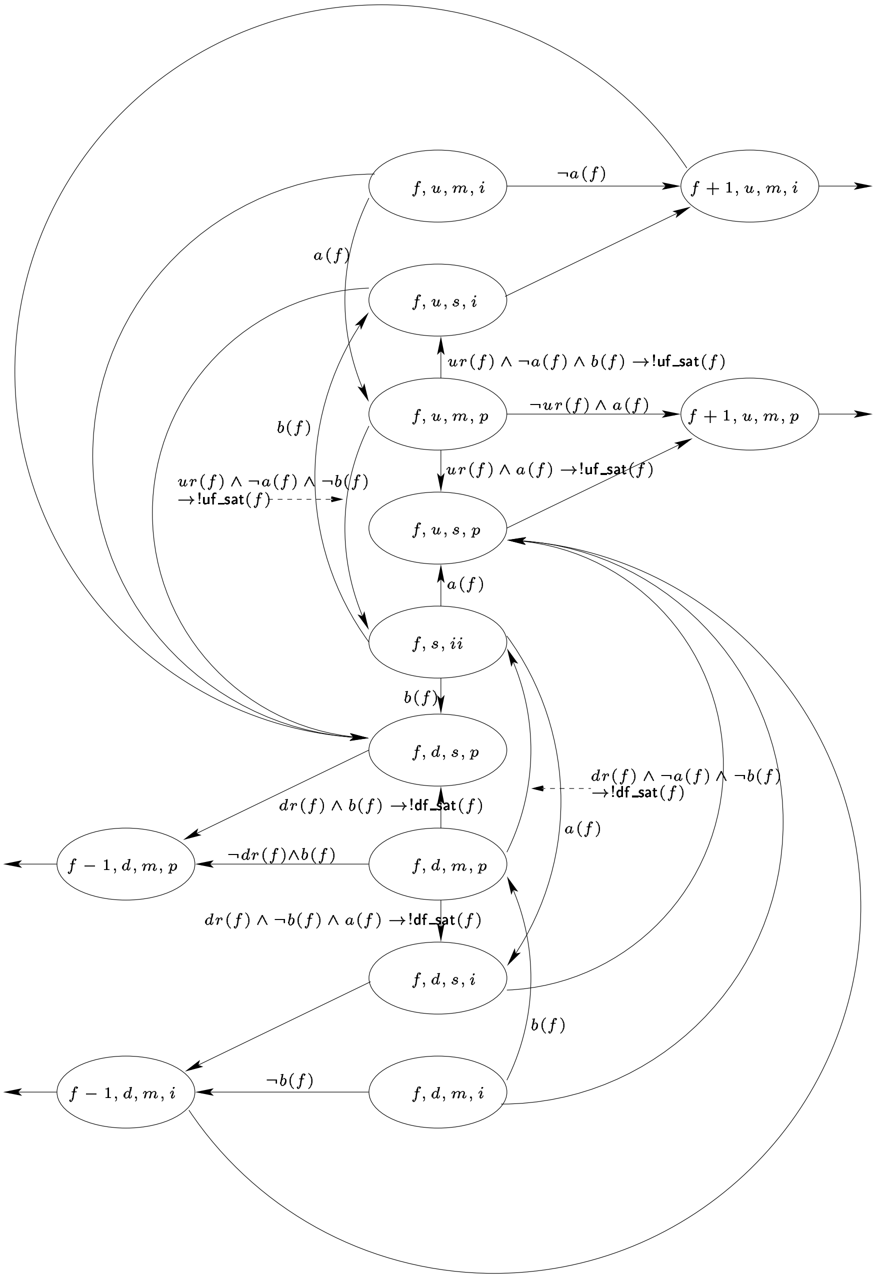

f, u, m, i
$CM.(f, u)$
$f+1, u, m, i$
$CM.(f+1, u)$
$CM.(f, d, s)$
f, u, s, i
$CM.(f+1, u)$
$CM.(f, u)$
f, u, m, p
$CM.(f, u)$
$f+1, u, m, p$
$CM.(f+1, u)$
$CM.(f, u)$
$CM.(f, u)$
$CM.(f+1, u)$
$CB.(f, u, \_, p)$
f, u, s, p
$CB.(f, s, ii) \vee CB.(f, u, \_, p)$
f, s, ii
$CB.(f, s, ii) \vee CB.(f, d, \_, p)$
f, d, s, p
$CM.(f+1, d)$
$CM.(f, d)$
$CM.(f, d)$
$CB.(f, d, \_, p)$
$CM.(f+1, d)$
$f-1, d, m, p$
$CM.(f, d)$
f, d, m, p
$CM.(f, d)$
f, d, s, i
$CM.(f+1, d)$
$CM.(f, u, s)$
$CM.(f+1, d)$
$f-1, d, m, i$
$CM.(f, d)$
f, d, m, i

Figure 4: CP_c



f, u, m, i
¬a(f)
f + 1, u, m, i
a(f)
f, u, s, i
ur(f) ∧ ¬a(f) ∧ b(f) →!uf_sat(f)
b(f)
f, u, m, p
¬ur(f) ∧ a(f)
f + 1, u, m, p
ur(f) ∧ ¬a(f) ∧ ¬b(f)
→!uf_sat(f)
ur(f) ∧ a(f) →!uf_sat(f)
f, u, s, p
a(f)
f, s, ii
b(f)
f, d, s, p
dr(f) ∧ ¬a(f) ∧ ¬b(f)
→!df_sat(f)
dr(f) ∧ b(f) →!df_sat(f)
a(f)
f − 1, d, m, p
¬dr(f)∧b(f)
f, d, m, p
dr(f) ∧ ¬b(f) ∧ a(f) →!df_sat(f)
f, d, s, i
b(f)
f − 1, d, m, i
¬b(f)
f, d, m, i

Figure 5: CB_b

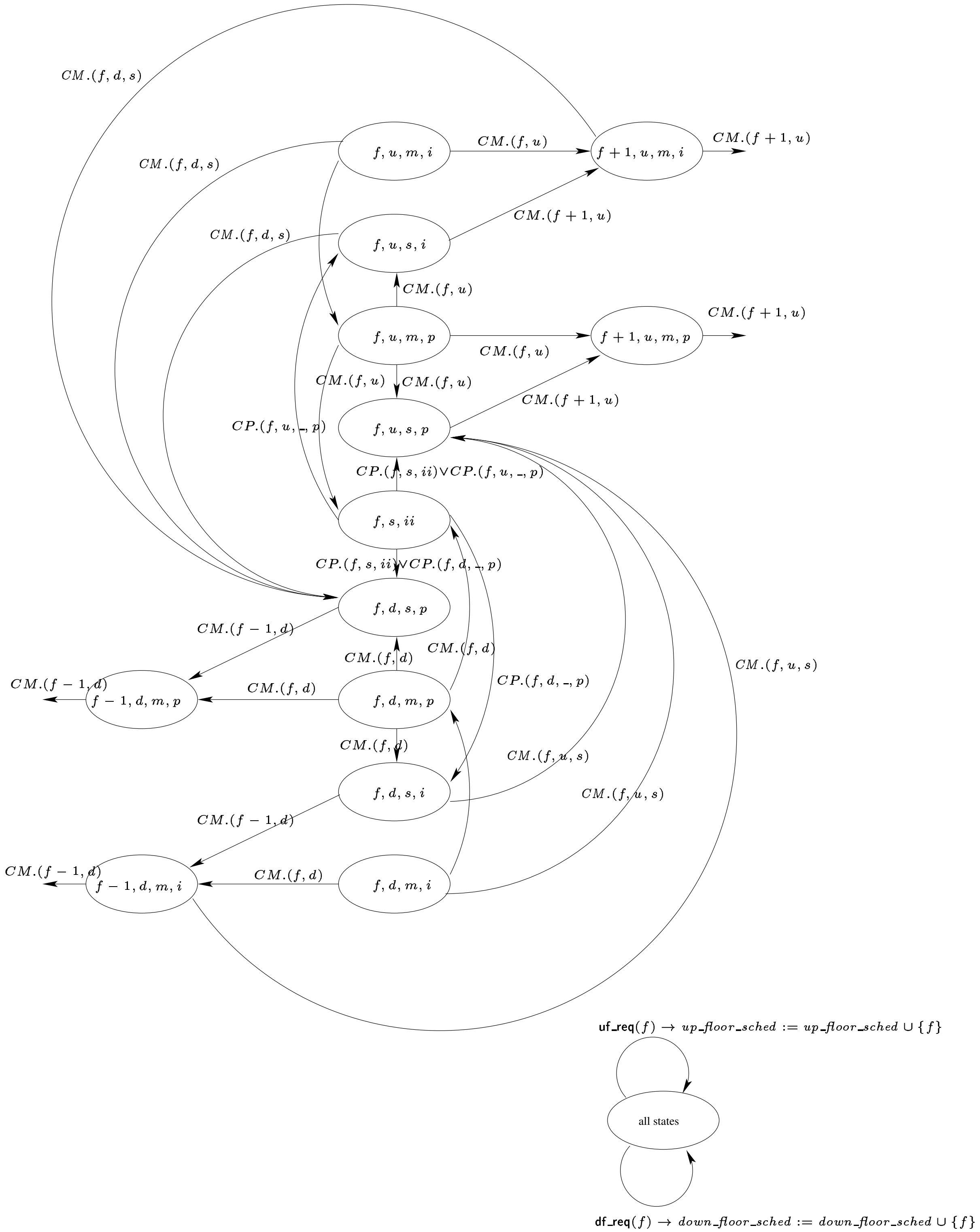

CM.(f, d, s)
CM.(f, d, s)
CM.(f, d, s)
f, u, m, i
CM.(f, u)
f + 1, u, m, i
CM.(f + 1, u)
f, u, s, i
CM.(f + 1, u)
CM.(f, u)
f, u, m, p
CM.(f, u)
f + 1, u, m, p
CM.(f + 1, u)
CM.(f, u)
CM.(f, u)
f, u, s, p
CM.(f + 1, u)
CP.(f, u, _, p)
CP.(f, s, ii)∨CP.(f, u, _, p)
f, s, ii
CP.(f, s, ii)∨CP.(f, d, _, p)
f, d, s, p
CM.(f − 1, d)
CM.(f, d)
CM.(f, d)
CM.(f, u, s)
CM.(f − 1, d)
f − 1, d, m, p
CM.(f, d)
f, d, m, p
CP.(f, d, _, p)
CM.(f, d)
CM.(f, u, s)
f, d, s, i
CM.(f, u, s)
CM.(f − 1, d)
CM.(f − 1, d)
f − 1, d, m, i
CM.(f, d)
f, d, m, i
uf_req(f) → up_floor_sched := up_floor_sched ∪ {f}
all states
df_req(f) → down_floor_sched := down_floor_sched ∪ {f}

Figure 6: CB_c

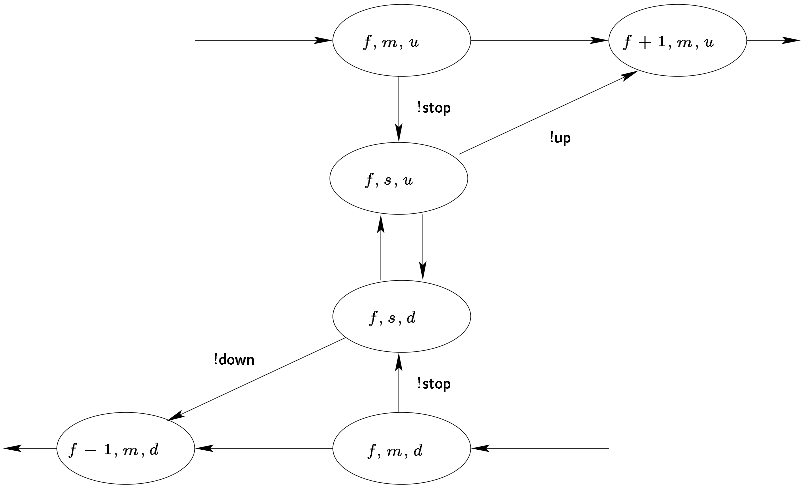


Figure 7: CM_m

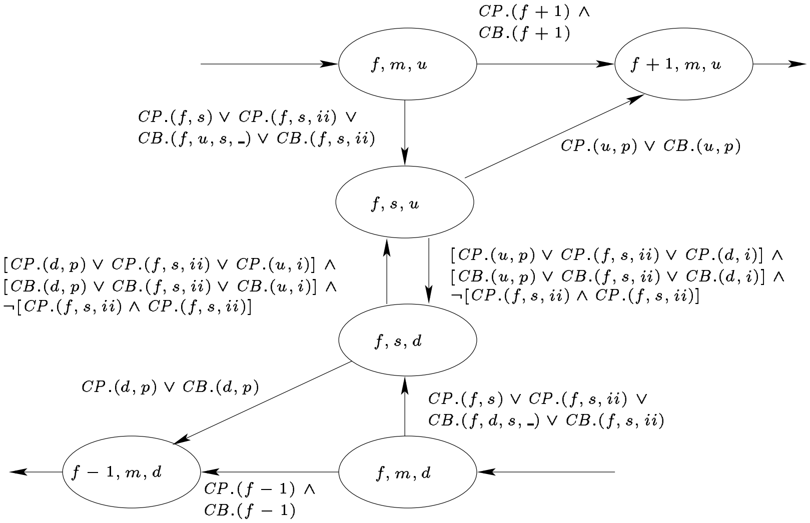

$CP.(f+1) \wedge$
$CB.(f+1)$
f, m, u
$f+1, m, u$
$CP.(f,s) \vee CP.(f,s,ii) \vee$
$CB.(f,u,s,\_) \vee CB.(f,s,ii)$
$CP.(u,p) \vee CB.(u,p)$
f, s, u
$[CP.(d,p) \vee CP.(f,s,ii) \vee CP.(u,i)] \wedge$
$[CB.(d,p) \vee CB.(f,s,ii) \vee CB.(u,i)] \wedge$
$\neg[CP.(f,s,ii) \wedge CP.(f,s,ii)]$
$[CP.(u,p) \vee CP.(f,s,ii) \vee CP.(d,i)] \wedge$
$[CB.(u,p) \vee CB.(f,s,ii) \vee CB.(d,i)] \wedge$
$\neg[CP.(f,s,ii) \wedge CP.(f,s,ii)]$
f, s, d
$CP.(d,p) \vee CB.(d,p)$
$CP.(f,s) \vee CP.(f,s,ii) \vee$
$CB.(f,d,s,\_) \vee CB.(f,s,ii)$
$f-1, m, d$
f, m, d
$CP.(f-1) \wedge$
$CB.(f-1)$

Figure 8: CM_c

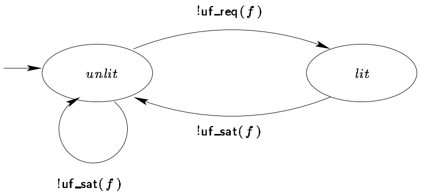


Figure 11: Up Floor Buttons

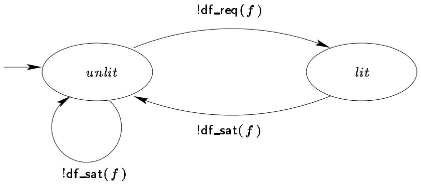


Figure 12: Down Floor Buttons

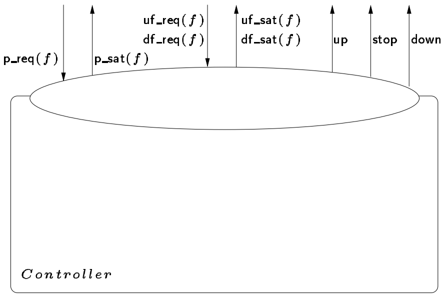


Figure 13: Centralized interface

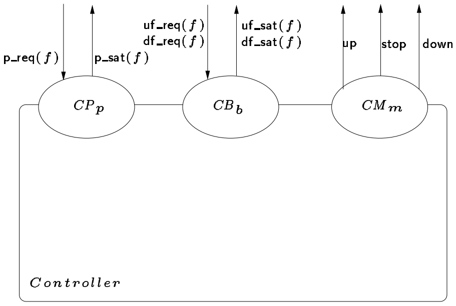


Figure 14: Separate interfaces

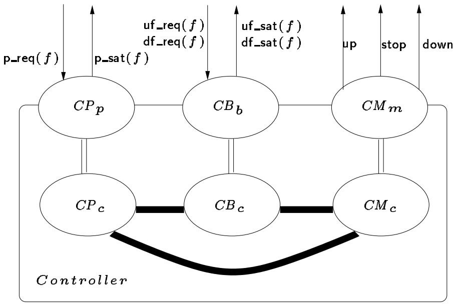


Figure 15: Split External-Internal interfaces

3.6 Verification of Behavioral Properties without State-explosion

Our interconnection scheme enables the separation of the state spaces of the various components. Instead of constructing the global product automaton of all the components in the elevator system, we only construct the pair machines $CP_p \parallel_m P$, $CB_b \parallel_m B$, $CM_m \parallel_m M$, and the "internal shared memory" composition $CP_c \parallel_i CB_c \parallel_i CM_c$.

We can then verify behavioral properties of each of these machines in isolation, and then combine these properties deductively to obtain properties of the overall system.

The $CP_p \parallel_m P$ pair-machine is very simple: it just records panel requests into *panel_sched*, and removes requests from *panel_sched* when the appropriate floor is visited. It also changes the state of CP as appropriate to record the current state of the panel requests, in particular, it manages the transitions between states with $CP.request = p$, states with $CP.request = i$, and states of the form $[f, s, ii]$.

Likewise the $CB_b \parallel_m B$ pair-machine records floor button requests into *up_floor_sched* and *down_floor_sched*, and removes them when the appropriate floor is visited with the elevator moving in the right direction, and also manages the $CB.requests$ bit.

We can therefore easily verify the following properties.

$CP_p \parallel_m P$ satisfies: (1) If a p_req(f) event occurs, then $f \in$ *panel_sched* subsequently remains true until $CP.floor = f$ holds, if ever.

$CB_b \parallel_m B$ satisfies: (1) If a uf_req(f) event occurs, then $f \in$ *up_floor_sched* subsequently remains true until $CB.floor = f \wedge CB.dir = u$ holds, if ever, and (2) If a df_req(f) event occurs, then $f \in$ *down_floor_sched* subsequently remains true until $CB.floor = f \wedge CB.dir = d$ holds, if ever.

$CM_m \parallel_m M$ satisfies: (1) Between an occurrence of event !up and a subsequent occurrence of !down, there is an occurrence of !stop, and (2) Between an occurrence of event !down and a subsequent occurrence of !up, there is an occurrence of !stop.

The $CP_c \parallel_i CB_c \parallel_i CM_c$ machine is more complex, since it in effect performs the coordination of the elevator. However, it has few states. Each of CP_c, CB_c, and CM_c has $9F$ states. Furthermore, the coordination between these three automata ensures that they are never more than one floor apart. Hence, the number of states in the product $CP_c \parallel_i CB_c \parallel_i CM_c$ cannot exceed $(2 \times 9)^3 F$, i.e., $5832F$. In practice, F does not exceed 200, giving an upper bound of 1166400 states, well within the reach of current automatic verification methods such as model checking [4]. We can therefore verify that $CP_c \parallel_i CB_c \parallel_i CM_c$ satisfies: (1) if $CP.floor = f$ holds, then eventually $CM.floor = f$ holds, (2) if $CB.floor = f \wedge CB.dir = u$, then eventually $CM.floor = f \wedge CM.dir = u$, and (3) if $CB.floor = f \wedge CB.dir = d$, then eventually $CM.floor = f \wedge CM.dir = d$.

We now verify the safety and liveness requirements given in Section 3.1. Properties (1) and (2) of $CM_m \parallel_m M$ immediately gives us safety. We establish liveness as follows. For panel requests, property (1) of $CP_p \parallel_m P$ together with property (1) of $CP_c \parallel_i CB_c \parallel_i CM_c$ gives us: if a $\mathsf{p\_req}(f)$ event occurs, then eventually $CM.floor = f$ holds. Hence all panel requests are eventually satisfied. For up button floor requests, property (1) of $CB_b \parallel_m B$ together with property (2) of $CP_c \parallel_i CB_c \parallel_i CM_c$ gives us: if a $\mathsf{uf\_req}(f)$ event occurs, then eventually $CM.floor = f \wedge CM.dir = u$ Hence all up floor button requests are eventually satisfied. For down button floor requests, property (1) of $CB_b \parallel_m B$ together with property (3) of $CP_c \parallel_i CB_c \parallel_i CM_c$ gives us: if a $\mathsf{df\_req}(f)$ event occurs, then eventually $CM.floor = f \wedge CM.dir = d$ Hence all down floor button requests are eventually satisfied. Having shown that all requests are eventually satisfied, we have established liveness.

Our example can be extended to several elevators by introducing interfaces between controllers which coordinate the servicing of floor requests, so that only one elevator services any given floor request. We omit the details.

4. DISCUSSION AND RELATED WORK

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 apply our approach at varying degrees of granularity. For example, we could have modeled each button as a separate component, with its own interfaces. We chose not to do this, and collected all panel buttons (floor buttons) into a single component, because the behavior of the buttons is so simple, that even though the button component has a large number of states ($O(2^F)$), it nevertheless is very easy to analyze.

By contrast, a brute-force approach in which the product of all components in the system was first generated, and then verified by model checking or reachability analysis, would have generated a global state space with at least $2F2^{3F}$, since there are three schedules (*panel_sched*, *up_floor_sched*, and *down_floor_sched*), and a centralized controller would require at least $2F$ states, to record the current floor and the direction of motion.

For a building with 200 floors, our approach requires verifying properties of a state space with at most 1166400 states, which is within reach of current methods, whereas a brute-force approach would require checking a state space of size $400 * 2^{600}$, or approximately 10^{180} states, which is clearly impractical.

Vanderperren and Wydaeghe [32, 28, 33, 27, 26] 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.

Alur and Henzinger [5] 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.

Component generators [18, 19, 20], synthesize a system of components from a concise specification. A combination of these techniques can form the basis for a method for the construction of large component-based software systems. The interactions amongst components can be checked pairwise, thus avoiding state explosion.

5. CONCLUSION

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

We implemented our system in the IOA toolset and language (see `http://theory.lcs.mit.edu/tds/ioa/`). A technical report presenting this implementation is available at `http://www.ccs.neu.edu/home/lorenz/papers/reports/NU-CCS-03-02.html`

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 woth the aid of theorem provers. One porposed approach to making formal methods economical is that of automatic model checking [4]: the state space of the system is mechanically generated and then exhaustively explored to verify the desired behavioral properites. Unfortunately, the number of global states is exponential in the number of components. This state-explosion problem is the main impediment to the succesfull 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 succesfully to the two-phase committ problem [1] and the dining and drinking philosophers problems [3].

Large scale component-based systems are widely acknowledged as a promising approach to constructing large-scale complex software systems. A key requirement of an succesful 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.

6. 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. 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.

[3] 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.

[4] 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. Extended abstract in Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages.

[5] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2001.

[6] Don S. Batory. Product-line architectures, aspects, and reuse (tutorial session). In *Proceedings of the 22nd International Conference on Software Engineering*, page 832, Limerick, Ireland, June 4-11 2000. ICSE 2000, IEEE Computer Society.

[7] Jan Bosch. Software product lines: Organizational alternatives. In ICSE 2001 [12], pages 91–100.

[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[9] E. A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Models and Semantics.* The MIT Press/Elsevier, Cambridge, Mass., 1990.

[10] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2:241 – 266, 1982.

[11] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001.

[12] ICSE 2001. *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, May 12-19 2001. IEEE Computer Society.

[13] W. Vanderperren and B. Wydaeghe. Towards a new component composition process. In *In Proceedings of ECBS 2001*, Apr. 2001.

[14] BeanBox. JavaSoft. `http://www.javasoft.com/-beans/software/beanbox.html`.

[15] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors. *Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering: Component Certifiation and System Prediction*, Toronto, Canada, May 12-19 2001. ICSE 2001, IEEE Computer Society.

[16] JavaStudio 1.0. JavaSoft, Mar. 1999. `http://java.sun.com/studio`.

[17] JBuilder. Inprise Borland. `http://www.borland.com/techpubs/jbuilder/`.

[18] D. H. Lorenz and J. Vlissides. Automated architectural transformation: Objects to components. Technical Report NU-CCS-00-01, College of Computer Science, Northeastern University, Boston, MA 02115, Apr. 2000. `http://www.ccs.neu.edu/home/lorenz/-papers/reports/NU-CCS-00-01.html`.

[19] David H. Lorenz and John Vlissides. Component design: Beyond object-oriented design. Technical Report NU-CCS-00-03, College of Computer Science, Northeastern University, Boston, MA 02115, November 2000. `http://www.ccs.neu.edu/home/lorenz/-papers/reports/NU-CCS-00-03.html`.

[20] D. H. Lorenz and J. Vlissides. Designing components versus objects: A transformational approach. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 253–262, Toronto, Canada, May 12-19 2001. ICSE 2001, IEEE Computer Society.

[21] Erkki Mäkinen and Tarja Systä. MAS - an interactive synthesizer to support behavioral modeling in uml. In ICSE 2001 [12], pages 15–24.

[22] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communication of the ACM*, 15(12):1059–1062, 1972.

[23] PowerJ 2.5. Sybase. `http://www.sybase.com/products/powerj/`.

[24] R. E. K. Stirewalt and L. K. Dillon. A component-based approach to building formal analysis tools. In *Proceedings of the 23rd international conference on Software engineering*, pages 167–176. IEEE Computer Society, 2001.

[25] C. Szyperski. *Component-Oriented Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1997.

[26] 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.

[27] W. Vanderperren and B. Wydaeghe. Separating concerns in a high-level component-based context. In EasyComp Workshop at ETAPS 2002, April 2002.

[28] W. Vanderperren and B. Wydaeghe. Towards a new component composition process. In Proceedings of ECBS 2001, April 2001.

[29] VisualAge for Java. IBM. `http://www.ibm.com/software/ad/vajava/`.

[30] VisualCafé. Symantec. `http://cafe.symantec.com/`.

[31] K. C. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. Software Engineering. Addison-Wesley, 2001.

[32] B. Wydaeghe. PACOSUITE: Component composition based on composition patterns and usage scenarios. PhD Thesis.

[33] B. Wydaeghe and W. Vandeperren. Visual component composition using composition patterns. In Proceedings of Tools 2001, July 2001.