Runtime Validation of Behavioural Contracts for Component Software

Yan Jin and Jun Han  
Faculty of ICT, Swinburne University of Technology, Hawthorn, VIC 3122, Australia  
{yjin, jhan}@ict.swin.edu.au

Abstract

In component software, the independence of components is achieved by separating their interfaces from implementations. The interface definition of a component serves as the contract with its neighbouring components regarding the use of its services. In general, such a contract should cover issues beyond interface signatures, such as service functionality, usage and quality. The Interface Definition Languages (IDLs) used by commercial middleware such as CORBA, however, lack mechanisms for capturing such semantic characteristics. In this paper, we introduce a framework and associated techniques that augment commercial IDLs with behavioural contract specifications and validate at runtime component interactions against such contracts. The behavioural contract of a component describes occurrence or sequencing constraints on its interactions with the environment. The validation of such constraints is achieved by intercepting runtime interactions between components and validating them against the finite state automata that semantically represent the constraints. The validation provides a useful tool for testing whether the component services are used properly and whether the component fulfils its behavioural obligations in a distributed system.

1. Introduction

A key feature of component-based software engineering is that it allows constructing an application using prefabricated pieces of software (or components) developed independently by third-parties so as to reduce costs and improve quality. In this construction process, it is essential to ensure that individual components can in fact interoperate to achieve the desired functionality and quality of the system.

From the system designer’s perspective, the normal expectation or assumption for using a component is that its internal implementation has already been extensively tested and it should have a high degree of reliability. Hence the most significant issues relating to the use of a component are whether it provides its services as the developer promises and whether it is correctly used in the system (i.e. all its assumptions are met) so that it can function properly.

In general, a component provides its services through interactions with the other components, e.g. message exchanges and operation invocations. The proper functioning of a component in a system depends on whether the other components behave as it expected and vice versa. The assumptions and guarantees from both the component and system perspectives thus form a behavioural contract between them. In ensuring the correct functioning of a system, it is important to capture the behavioural contract for each component and ensure that they are not violated at runtime. Runtime monitoring of the contracts can provide valuable information on system execution traces for identifying the liable party when a violation is detected.

In this paper, we focus on the precise specification of behavioural contracts from the components’ perspective and their validation at runtime. Such contract specifications are important because they accurately reflect how components are actually implemented and enable the formal analysis of systems made up of third-party components. The contracts can be further specialised by the system designer to reflect application-specific constraints.

Our contract specification adopts a lightweight approach presented in [19] and extends it with the property specification pattern system (SPS) from [13]. A behavioural contract is defined as the conjunction of interaction constraints (ICs) and can be published together with the component IDL description. Each IC states an occurrence or sequencing rule on certain events or operation invocations. To make it easy for software practitioners to specify ICs, we provide temporal operators for common property patterns and mechanisms to define conditions on operation parameters. Typically, such specification tasks are performed by the component developer who is able to validate the constraints against the component implementation in the development process.

We develop a framework and associated techniques that support the automatic validation of runtime component interactions against the specified ICs. This framework employs finite state automata (FSAs) as semantic representations of ICs, uses interceptors for tracking runtime compo-
nent interactions, and develops mechanisms for managing dynamically created constraint FSAs as well as for conducting automatic conformance checking of component interactions against ICs. The validation becomes a useful tool for testing whether or not the component services are used properly and whether or not the component fulfils its behavioural obligations in a distributed system. So far this framework has been implemented in the CORBA context. It is nevertheless applicable to other middleware platforms.

The rest of the paper is organized as follows. Section 2 highlights the need for component behavioural contracts and gives a brief discussion on existing solutions to their specification and the remaining problems. Following an overview of our IC-based contract specification approach in Section 3, Section 4 introduces our runtime contract validation framework. Section 5 discusses related work before in Section 6.

2. Background

A motivating example. The use of commercial IDLs, including CORBA IDL, to describe component interfaces enables the enforcement of signature compatibility between components, such as operations, parameters and data types. This however fails to capture dynamic properties of components such as behavioural contracts. Let us consider an example auctioneer component simplified from [19] and originally taken from [7]. The auctioneer communicates with a number of bidders in a distributed auction system. It is able to accept registrations from the bidders and hold auctions among registered bidders. It provides two operations for registration and unregistration and requires a bidder to provide three operations for bidding inquiry, bidding settlement, and settlement announcement. Figure 1 shows the IDL definitions for auctioneers and bidders, where refNo stands for the unique reference number to an auction.

As shown, the IDL definition does not convey semantic information about the auctioneer. In order for the system designer to use this component properly and enforce system-wide interoperability, additional information, e.g. its behavioural contract (or interaction protocol) with the environment, is needed. This may describe how it provides its services, how its services can be utilised, and what are the obligations of the clients in using the services. For example, which bidder(s) will the auctioneer select to send bidding inquiries? Will an item be sold as long as there exists a bid? Will abnormal behaviour be possible, e.g. selling an item to a bidder who did not bid?

Requirements for contract specification. To address this, a systematic but practical approach to describing component behavioural contracts is needed, which should consider the following issues:

- The targeted user is software practitioners. They typically lack training in formal specification languages and methods. The contract description language, process and strategies should therefore be, while sufficiently expressive, as simple and intuitive as possible. Consequently, the practitioners can easily ensure the proper application of the approach.

- The exposed contract descriptions must be unambiguous and amenable for formal reasoning. This is important for developing automated reasoning tools.

- As components are provided for use in diverse systems, component descriptions should not depend on specific contextual system architectures. Likewise, they should not depend on the internal implementation.

- Since software is always evolving, component contract descriptions should be easily modifiable to reflect the implementation changes. It should be easy for the system designer to figure out the changes by comparing the old and new descriptions in order to change the system integration architecture accordingly.

Existing solutions. In practice, informal documentation is often used, which describes component behavioural contracts in a natural language. Although intuitive, this often fails to produce a consistent interpretation due to the ambiguity of natural languages.

Many formal specification approaches have therefore been proposed in the literature, e.g. [1, 2, 4, 5, 6, 7, 8, 10, 27, 28, 29]. They rely on the use of formal specification languages and are able to produce precise descriptions of component behavioural contracts. For example, [8, 12, 28, 29] use finite state machines (FSMs), [2, 10] interface automata, [1, 5, 7] process algebra, [27] regular expressions, [4] Petri nets, and [6] description logics. The benefit of using a well-developed formalism is that many analysis techniques are already available. A major weakness is that applying such an approach requires the user to have a sound knowledge of the underlying formalisms and methodologies. Their adoption is thus limited among practitioners who usually do not have the required expertise.
Some existing approaches such as [1, 2, 5, 27] build on ADLs. They mainly concern with the overall system architecture and behavioural properties. The component behaviour is often specified based on knowledge of the architecture design, e.g. who and how other components are connected with a component. As such, unless certain restrictions on the system architecture are imposed, it can be difficult to describe components from the components’ perspective for third-party use.

Other approaches such as [4, 7, 10, 12, 29] allow specification of behavioural contracts of components for third-party use. They associate each component with a single protocol model which manifests all the valid interaction scenarios. As a result, they do not scale well as the behavioural contract becomes complex. Due to interleavings of interaction events, it can be difficult for third-parties to understand a complex contract. Once published, a change to the contract implies the creation of a new protocol model. This in turn implies the system analysis based on the earlier model may need to be conducted again.

Approaches such as [8, 28] suggest the use of multiple models to represent the behavioural contract, in order to provide flexibility to the specification. However, they did not address important issues such as capturing the sequencing constraints between events in different models and the effect of different parameter values on the event sequencing.

In summary, the following issues are not addressed adequately for existing approaches to be of practical use: (1) the use of formalisms and methods of which ordinary software engineers do not have firm grasp; (2) the use of knowledge beyond the component itself; (3) insufficient support for incremental evolution of the behavioural contract.

3. Specifying behavioural contracts

To address the above-mentioned issues, we proposed in [19] a specification approach which defines the component behavioural contract as the conjunction of interaction constraints (ICs). Each of the ICs states an occurrence or sequencing rule on message exchanges with the component. As only a few operations are involved, the specification of an IC is much easier and the effort to understand it is minimal. Also, to facilitate the use by software engineers, intuitive temporal operators of interaction scenarios are provided, which hide the “intricate” formality from the user.

In this paper, we bring in the specification patterns system (SPS) proposed by Dywer et al. in [13] for defining temporal operators and consequently extend [19] with expressiveness and the ability to cope with complex sequencing constraints on component interactions. SPS was originally developed to assist practitioners to formally specify system properties. It was shown in [14] that SPS caters for a majority of system properties.

```plaintext
component Auctioneer {  
  provides IAuctioneer;  
  requires IBidder;  
  interaction-contract {  
    peer-level {  
      wannaBid exists only after register                  (1)  
      wannaBid precedes youGotIt                            (2)  
      where wannaBid.refNo = youGotIt.refNo &
                                wannaBid.result = true;  
    }  
    component-level {  
      wannaBid leads to itemSold                               (3)  
      where wannaBid.refNo = itemSold.refNo &
                                wannaBid.result = true;  
    }  
  }  
}
```

Figure 2. Example interaction constraints

Specifically, SPS identifies patterns for restricting the occurrences of individual events and the order (or sequencing) of occurrences between different events. Patterns of the first kind include universality, absence, existence, and bounded existence. For example, a bounded existence property “e1 exists at most once” indicates that event e1 can occur at most once. Patterns of the second kind include precedence, response, precedence chain, and response chain. For instance, given two events e1 and e2, a precedence property “e1 precedes e2” states that e1 must occur at least once before any occurrence of e2. One may think e1 enables e2. A response property “e1 leads to e2” states that an occurrence of e1 must eventually be followed by an occurrence of e2. In essence, this specifies a cause-effect relationship between e1 and e2. In general, to handle more complex properties, SPS patterns can be nested and be associated with scopes such as global, before e3, after e3, between e3 and e4 and after e3 until e4, where e3 and e4 are distinct events. The reader is referred to [13, 14] for detailed descriptions of SPS. To facilitate later presentation, we define “e2 exists only after e1 until e3” as a shortcut for two properties: e1 precedes e2 and e2 is absent after e3 until e1.

Figure 2 presents three example IC specifications for the auctioneer component. Each of them applies a pattern and a scope to a number of operations, and defines a sequencing rule between certain invocations of these operations. The specific invocations of interest and the relationship between their parameter values and results are specified by the associated where condition. As shown, ICs are declared at two levels: peer-level and component-level. Peer-level ICs are used to constrain the interactions with individual neighbouring components, while component-level ICs do not distinguish between the neighbours.

In particular, the first constraint is at the peer level and thus applies to each individual bidder. It states the relative sequencing between invocations of register, wannaBid...
and unsubscribe from/to a same bidder, regardless of the parameter values. Essentially, it requires the auctioneer only query registered bidders for their bidding interests in a particular auction. Note that wannaBid may be invoked several times or not at all between the other two operations.

The second IC is also at the peer level and requires that, if the youGotIt operation of a bidder is invoked regarding an auction refNo, this must be preceded by a bid of that bidder for refNo (i.e., wannaBid returns true). That is, an item being auctioned is only sold to a bidder who bid for it.

The third IC is at the component level and states the causality between wannaBid and itemSold, i.e., an item being auctioned will be sold if a bid is made. Note that it enforces no identity correspondence between the two callees.

The above ICs present a partial set of rules governing the interactions with the auctioneer. It is, however, easy to add more ICs so as to make a stronger interaction contract. For instance, the auctioneer is required not to sell an item to more than one bidder at an auction. One can add a component-level constraint ‘‘youGotIt exists at most once wrt refNo’’, stating that, for any given auction refNo, youGotIt can be invoked at most once. One can also require that the auctioneer only invoke the youGotIt and itemSold operations of registered bidders, by specifying a peer-level IC similar to the first one in Figure 2.

As seen above, the behavioural contract specification adopts an incremental approach. The separation of concerns using interaction constraints facilitates the understanding of the specification and the proper use of components, and enables incremental reasoning of interoperability. As shown later, it is possible at runtime to selectively enable and disable ICs for efficient contract validation. Further, a change to the contract can be made incrementally. This helps minimise the impact of contract change on the user side.

4. Runtime contract validation

Explicit specification of component interaction constraints helps the component developer and the user to implement and use a component properly. Whether or not the component services are actually used properly at run-time is a different question. Validation or testing is often required. In this section, we introduce a framework that allows us to validate run-time interactions of components against their defined interaction constraints in a distributed system. It can be used to complement formal verification techniques in order to provide additional confidence in the correctness of the system design and implementation.

4.1. Validation framework

Our validation framework adopts a monitoring approach to observe and validate component interactions. The run-

![Figure 3. Validation framework](image)

time monitoring and validation is fully automated in the CORBA context by a prototype tool called CIPV (Component Interaction Property Validator), and is achieved by

- employing CORBA portable interceptors (PIs) [26] to identify and intercept the run-time communications of components needed for validation;
- handling explicitly values of named parameters in IC specifications and representing the resultant concrete ICs using finite state automata (FSAs) for easy processing in the validation;
- constructing and removing constraint FSAs dynamically (upon intercepting events of interest) to cope with operation parameters with large or infinite domains, and systems with dynamically created components;
- checking the intercepted run-time interactions against the currently active constraint FSAs, and reporting violations (if any).

Figure 3 shows the overall monitoring architecture. The monitored components (or CORBA objects) are wrapped by CORBA PIs. These interceptors are programmed to observe the incoming and outgoing operation invocations as well as the creation and deletion of components, and to forward them to the CIPV validation module.

The validation module consists of five parts: an operator library (OL), an Event Dispatcher (ED), Component Specification Managers (CSMs) (one for each type of monitored components), Component Validation Managers (CVMs) (one for each monitored component), and IC validators (ICVs) (one for each concrete interaction constraint.
4.2. Semantic representation for ICs

The IC specification approach presented above is designed for intuitive use. However, to support run-time validation, it requires a precise semantics for ICs. The semantics of SPS patterns defined by [13] builds on temporal logics or regular expressions and thus is not easy to use in the automatic run-time conformance checking. Therefore, we choose finite state automata (FSAs) as the semantic representation of patterns and interaction constraints.

In general, each pattern operator or template has a corresponding FSA representation where arc labels are formal event parameters. As an example, Figure 4 presents three concrete component (if applicable), and values to the named parameters. Generally, a FSA transits from the source state to the target state by taking an event in a labelling set of an arc. A finite sequence of events is valid if taking the events sequentially leads the FSA to an accepting state (denoted by a double-circle). In Figure 4, certain abbreviations are used. For example, $W_{a_{1}\rightarrow b_{1}}$ denotes the set of all call and return events of $wannaBid$ between $a_{1}$ and a bidder $b_{1}$. $R_{refNo=101}$ is the set of return events from $a_{1}$ to $b_{1}$, and $I_{refNo=101}$ is the set of $itemSold$ call events from $a_{1}$ to any component with $refNo = 101$, where “$*$” means “do not care”. Also, $A_{a_{1}}$ denotes the universe of events of $a_{1}$, $O_{a_{1}} \subseteq A_{a_{1}}$ the set of all events not explicitly mentioned in the figure, and “!$\rightarrow$” set deduction.

Figure 4(a) shows the resultant concrete FSA for $a_{1}$ when the first peer-level IC specification in Figure 2 is applied to a bidder $b_{1}$. All $wannaBid$ events are only acceptable at the right state but rejected at the left state where $b_{1}$ is unregistered. Figure 4(b) shows the FSA obtained when applying the second peer-level IC specification in Figure 2 to $b_{1}$ and auction 101. Calls to youGotIt regarding auction 101 are inhibited at the initial state and acceptable only after a bid from $b_{1}$ is received. Figure 4(c) shows the corresponding FSA when the component-level IC in Figure 2 applies to auction 101. A call to $itemSold$ is required after any bid and before $a_{1}$ terminates in order for the FSA to move from the right state to the accepting state and consequently produce a valid event sequence.

4.3. Dynamic management for ICs

As noted above, we build a FSA for each combination of IC specification, component instance, neighbour component (if applicable), named operation parameter and value. In general, it is not possible to statically build all FSAs for each component because certain parameters may range over large or infinite domains. It is also because components may be dynamically created and destroyed and consequently the identities and number of neighbouring components vary at different points of time. Hence the construction of FSAs has to be performed dynamically when the need arises. Likewise, FSAs can become obsolete at certain stages of system execution. For example, when a component is destroyed, not only all its associated FSAs but also the corresponding peer-level constraint FSAs of each of its neighbours are of no further use. Such FSAs should be removed from the validation space.

To handle dynamic creation and destruction of FSAs, CIPV employs a CVM for each component in the moni-
tored system. The manager maintains a database of concrete ICs that are in effect (at both peer and component levels). Upon receiving an intercepted event (or operation invocation), the CVM creates a new FSA and an ICV if (1) the event matches an operation in an IC specification, and (2) there exists no corresponding concrete IC in the database. As noted in Section 4.1, testing the first condition is done by querying the corresponding CSM. In constructing a FSA, the CVM loads from OL the formal FSA of the used temporal operator and actualises the formal event sets.

When a component is removed from the system, the CVM removes all its associated FSAs and ICVs, and notifies the CVM of every component that has communicated with this component. When notified, each neighbour CVM removes all its associated FSAs and ICVs, and no-longer forwards intercepted events (or operations invoked on the destroyed object) to the component to be destroyed.

Figure 5 presents a much shorter sequence of the past events, after a violation occurs and those of the peer component. It also presents a much shorter sequence of the past events, after filtering out irrelevant events from the outgoing arcs. For example, a wannaBid call is received at the left state of Figure 4(a).

As an example, suppose a₁ sends a bidding inquiry “wannaBid(102, bag, 50.0)” to b₁ after b₁ has unregistered itself. Then a violation to the FSA in Figure 4(a) will occur. A violation report detailed in Figure 6 will be presented to the system tester. It details the violated IC specification, the type and identity of the component where the violation occurs and those of the peer component. It also presents a much shorter sequence of the past events, after filtering out irrelevant events from O₁₁ (as in Figure 4(a)).

4.4. Checking interaction constraints

As noted, concrete interaction constraints are semantically represented as FSAs. When a component/object is created, its CVM is initialised. When an interaction event is captured by the interceptors attached to the object, it is forwarded to the CVM. The manager locates all the FSAs that concern the event, including the FSAs newly created in response to this event reception, and commands the corresponding ICVs to advance the FSAs using the event. In doing so, violations to the constraints can be determined. The algorithms involved are straightforward and are omitted here due to space limitation. In general, a constraint violation occurs in any of the following situations:

- The captured event is rejected at the current state of a FSA, viz. it does not appear in any labelling set of the outgoing arcs. For example, a wannaBid event is received at the left state of Figure 4(a).
- The captured event is a component or system destruction event and some active FSAs are not at an accepting state. For instance, a₁ terminates when the FSA in Figure 4(c) is at the right state.

4.5. Tool implementation

The CIPV tool is written in Java using the J2SE SDK v1.4.2. Its implementation primarily consists of two parts: CORBA-specific runtime interaction monitor and CORBA-independent interaction validator. The monitor implements interceptors using the CORBA Portable Interceptor libraries provided in J2SE and an ORB product “ORBacus 4.2.2” provided by IONA [18]. It runs at the component side and
incremental addition is intuitive and easy to follow. Also, its specification makes use of well-established property patterns for specifying constraints. As such, a behavioural contract description is given by a conjunction of occurrence and interleaving constraints on operation invocation events. It is the interplay between operation parameters and inter-leavings between operation invocations that make it easier for practitioners to apply.

As seen previously, our approach defines a component behavioural contract as the conjunction of occurrence and sequencing constraints on operation invocation events. It makes use of well-established property patterns for specifying the constraints. As such, a behavioural contract description is intuitive and easy to follow. Also, its specification is incremental, i.e., changes to the contract are made by easily adding and removing interaction constraints. These features are important for it to be of immediate practical use. They differ our approach from many existing specification approaches such as [1, 2, 4, 5, 6, 7, 8, 10, 12, 27, 28, 29].

Also related to our approach is the work based on Design by Contract [24] such as [3, 9, 21, 25]. The behavioural contracts of components (or classes) are specified in terms of class invariants, pre- and post-conditions of operation invocations. Although expressive, the contract specifications are interleaved with source code and tied with the implementation language such as Eiffel or Java [23]. This makes it difficult to validate the contracts by system designers when the source code of components is not available.

In the literature, there exist a considerable number of runtime monitoring approaches and tools which are able to determine whether the current execution of a system preserves the specified properties. A classification and comparison of those approaches is given in [11]. Many of the approaches, such as DynaMICs [15], Java-MaC [20], Jass [3], and Java PathExplorer [17], adopt an instrumentation approach and require access to the source code (or Java byte-code) of the component implementation in order to inject additional monitoring or validation code. In contrast, our runtime monitoring approach only makes use of interceptors for monitoring purposes, which are independent pieces of software from the component implementations. Such an interceptor-based monitoring approach was also taken by [7, 22]. Generally, it is easier to remove the interceptors than the instrumentation code from the tested system when needed. Furthermore, our work differs from the existing monitoring approaches, including [7, 22], in the way that behavioural properties are specified. None of them uses patterns for specifying system properties to be validated. As [14] shows, from practitioners’ point of view, property patterns (or operators in our approach) are easier to use than temporal logics, the property specification languages commonly used by existing monitoring approaches.

5. Related work

The work presented in this paper is a continuation of [16]. The approach has been developed and used in an industrial telecommunication system context. Compared to [16], our specification approach is more expressive due to (1) the ability to cater for operation parameters and inter-leavings between operation invocations and (2) the introduction of the property specification patterns from [13]. In contrast to the static mechanism in [16], our validation approach employs a dynamic FSA management mechanism to handle values with large or infinite domains and save memory consumption for FSAs that have not taken effect and that have been obsolete.

As seen previously, our approach defines a component behavioural contract as the conjunction of occurrence and sequencing constraints on operation invocation events. It makes use of well-established property patterns for specifying the constraints. As such, a behavioural contract description is intuitive and easy to follow. Also, its specification is incremental, i.e., changes to the contract are made by easily adding and removing interaction constraints. These features are important for it to be of immediate practical use. They differ our approach from many existing specification approaches such as [1, 2, 4, 5, 6, 7, 8, 10, 12, 27, 28, 29].

Also related to our approach is the work based on Design by Contract [24] such as [3, 9, 21, 25]. The behavioural contracts of components (or classes) are specified in terms of class invariants, pre- and post-conditions of operation invocations. Although expressive, the contract specifications are interleaved with source code and tied with the implementation language such as Eiffel or Java [23]. This makes it difficult to validate the contracts by system designers when the source code of components is not available.

In the literature, there exist a considerable number of runtime monitoring approaches and tools which are able to determine whether the current execution of a system preserves the specified properties. A classification and comparison of those approaches is given in [11]. Many of the approaches, such as DynaMICs [15], Java-MaC [20], Jass [3], and Java PathExplorer [17], adopt an instrumentation approach and require access to the source code (or Java byte-code) of the component implementation in order to inject additional monitoring or validation code. In contrast, our runtime monitoring approach only makes use of interceptors for monitoring purposes, which are independent pieces of software from the component implementations. Such an interceptor-based monitoring approach was also taken by [7, 22]. Generally, it is easier to remove the interceptors than the instrumentation code from the tested system when needed. Furthermore, our work differs from the existing monitoring approaches, including [7, 22], in the way that behavioural properties are specified. None of them uses patterns for specifying system properties to be validated. As [14] shows, from practitioners’ point of view, property patterns (or operators in our approach) are easier to use than temporal logics, the property specification languages commonly used by existing monitoring approaches.

6. Conclusion

Unambiguous descriptions of behavioural contracts from components’ perspective can accurately reflect how components are actually implemented. This is critical for ensuring the proper use of software components produced by third-parties or different teams in a distributed system and consequently the correct functioning of the system.

In this paper, we have presented a lightweight specification approach which enhances commercial DLs with interaction constraints. The constraints conjunctively determine the behavioural contract of a component. The suggested use of well-understood property patterns in specifying such constraints makes it easier for practitioners to apply.

We have also presented a framework that supports the monitoring and validation of runtime component interac-

---

**Figure 6. An example violation report**

The figure shows an example violation report that forwards the messages sent or received by the component to the validator. Note that the programmed interceptors are generic and independent of specific CORBA applications.

The runtime validator manages all CVMs for monitored components, implements the parsing of component IC specifications, and maintains the FSA semantics of patterns operators. A CVM is in charge of creating new FSAs in response to the capture of new events, managing all the FSAs for active ICs, and the automatic checking of intercepted events against these FSAs.

Further, the validator provides a window-based GUI for the system tester to view the interaction history of each component, manage the concrete ICs in effect, and examine the dynamic state transition of FSAs in a graphical form.
tions against the specified behavioural contracts. The key techniques include the use of interceptors for observing component interactions, the use of FSAs for representing interaction constraints, the use of a dynamic FSA management approach, and the fully automatic checking of FSAs by a prototype tool CIPV. In effect, this framework provides a useful tool for determining whether a component is used properly and whether it functions properly in a distributed CORBA application. This helps the designer ensure system-wide interoperability between components. We note that although this framework is implemented in the CORBA context, its CIPV validation module is independent of CORBA and can be reused with other middleware and service frameworks.

Acknowledgements. The authors would like to thank Andrew Russo, Phan Manh Tan, and Doan Thuy An for their effort in implementing the CIPV tool.

References