# Using the TrustME Tool Suite for Automatic Component Protocol Adaptation

Ralf Reussner[1], Iman Poernomo[1], and Heinz W. Schmidt[2]

[1] Distributed Systems Technology Center (DSTC) Pty Ltd
Monash University, Melbourne, Australia
{reussner|imanp}@dstc.com
[2] Center for Distributed Systems and Software Engineering (DSSE) ,
Monash University, Melbourne, Australia
hws@csse.monash.edu.au

**Abstract.** The deployment of component oriented software approaches gains increasing importance in the computational sciences. Not only the promised increase of reuse makes components attractive, but also the possibilities of integrating different stand-alone programs into a distributed application. Middleware platforms facilitate the development of distributed applications by providing services and infrastructure. Component developers can thus benefit from a common standard to shape components towards and application designers from using pre-fabricated software components and shared platform services. Although such platforms claim to achieve fast and flexible development of distributed systems, they fall short in key requirements to reliability and interoperability in loosely coupled distributed systems. For example, many interoperability errors remain undetected during development and the adaptation and integration of of third-party components still requires major effort and cost. Partly this problem can be alleviated by the use of formal approaches to automatic interoperability checks and component adaptation. Our Reliable Architecture Description Language (RADL) is aimed at precisely this problem. In this paper we present key aspects of RADL used to specify component-based, compositional views of distributed applications. RADL involves a rich component model, enabling protocol information to be contained in interfaces. We focus on protocol-based notions of interoperability and adaptation, important for the construction of distributed systems with loosely coupled components.
**Keywords:** component protocol specifications, automatic component adaptation, parameterised contracts, architectural description languages, distributed middleware platforms.

## 1 Introduction

The deployment of component oriented software approaches gains increasing importance in the computational sciences. Not only the promised increase of reuse makes components attractive, but also the possibilities of integrating different stand-alone programs into a distributed application. Distributed systems are often constructed using middleware platforms. For example, enterprise application

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 854–863, 2002.

development is increasingly aided by implementations of OMG's CORBA [16] and others, while scientific distributed application development is often based upon PVM or its successor HARNESS [13]. These middleware platforms can facilitate distributed programming, through the provision of services and infrastructure such as distributed name lookup, remote procedure calls, and transactions. In general, middleware supports these features through an underlying component interface model, in which a distributed architecture is decomposed into black-box software units, whose individual functionality is provided by interfaces consisting of method signatures that are understood by means of an informal API [20].

However, errors in design and deployment of distributed middleware-based systems can still arise. This can be partly alleviated by the use of formal, tool-based approaches to configuration of system components. Two feasible aids to configuration are: automatic interoperability checks – checking that components are correctly communicating – and automatic adaptation – enabling a component to correctly communicate with another. System design and implementation can benefit from automatic interoperability checks, because this leads to detection of configuration errors prior to final deployment. Also, automatic component adaptation leads to faster development and less error-prone configurations of interoperating components.

In the TrustME project our approach to these problems is based on a software architectural description language (ADL). We are developing tools that facilitate automatic interoperability checks and adaptation for distributed middleware-based systems. An ADL is used to specify a high-level, compositional view of a distributed software application, specifying how a system is to be composed from components, and how and when components communicate. ADLs usually come equipped with a component model that contains a state-transition style semantics, enabling behavioural analysis [11]. The TrustME ADL is called RADL, short for "Rich" or "Reliable" ADL, because it enriches standard architecture descriptions with interoperability protocol features aiming at reliable interoperability. In RADL a component protocol is considered the set of valid call sequences of component methods. RADL augments a simple component interface model, consisting of a method signature, with additional protocol information and uses architectural composition to associate different fragments of interface specifications to different portions of the overall architecture definition. This leads to a notion of component and system contracts and to parameterised contracts. Our tool, called *CoConut/J*-tool,[1] is able to perform a certain class of component protocol adaptations automatically. This adaptation is based upon parameterised contracts [18], a generalisation of classical software contracts [12].

Through focusing on protocols for checking and adaptation, our work has a strong bearing on concurrent distributed systems architecture and design. There, components communicate through message exchange and it is impor-

[1] The *CoConut/J*-tool was originally developed by the first author at the Universität Karlsruhe, Germany as part of the *CoConut/J*-project (Contracts for Components) under DFG-funding.

tant to model the valid sequences of message exchange, i.e., the protocol used for interoperation.

The paper is organised as follows. In section 2 we outline RADL and its component model. In section 3 we provide an overview of parameterised contracts and how they can be concretely applied to our component model defined within the ADL. An example of how our tool implements this work is presented in section 4. Section 5 discusses related work, and section 6 concludes.

## 2 RADL: An ADL for Component Protocols

The TrustME ADL[2] was originally described in [19], and has been extended further [17]. Like other ADLs, RADL provides a means of defining compositions of component-based systems, but, in contrast to most ADLs, it enables interoperability checking between interfaces of components in an architecture.[3]

The TrustME language decomposes a distributed middleware-based system into hierarchies of components, linked to each other by connections between their interfaces:

- *Components* are self-contained, coarse-grain computational entities, potentially hierarchically composed from other components.
- *Component interfaces* consist of a signature of *services* and protocol definitions. Interfaces are either *provided* or *required*. The former interfaces specify services that are offered by the component, while the latter describe methods that are required by the component.

Our components and interfaces are analogous to their namesakes in Darwin, to components and ports respectively in C2 and ACME, or to processes and ports in MetaH [11]. However, RADL differs from these other languages, in that, by providing a more detailed interface description, it enables automatic interoperability checks between interfaces (most other ADLs treat interfaces as simply a collection of service names).

### 2.1 Protocol descriptions using finite state machines

The protocol of a provides-interface (i.e., the *provides-protocol*) can be considered as a set of *valid* call sequences to that interface according to the interface signature. A valid call sequence is a call sequence which is actually supported by the component. For example, a file I/O component might provide `open`, `read`, `close` as services, where `open-read-close` is a valid call sequence, while `read-open` is

[2] TrustME is the product of collaborative research conducted in the DSTC and at Monash University.

[3] The ADL is heterogeneous, possessing a generic and extensible means of defining modelling constructs, and permitting various component interface models. In this paper, we outline the subset of TrustME relevant to describing component protocols within middleware-based architectures.

not. Analogously, the protocol of a requires-interface (i.e., the *requires-protocol*) is a superset of the calls sequences, by which a component calls external services.

In both cases, a protocol is considered as a set of service call sequences. We use finite state machines (FSMs) to specify interface protocols, because FSMs provide a commonly understandable formalism and compositionality and substitutability of components can be checked efficiently due to efficient algorithms to check the inclusion of FSMs. A FSM consists of states and state transitions. State transition diagrams, such as that of Fig. 2, denote FSMs, where state are represented by circles and transitions by arrows. There is one designated state, *start-state*, in which the component is after its creation. Each state is associated with a set of services that are callable in that state. Service calls are given denoted by state transitions, because calls change the state – usually, other services are callable after the call, while others, callable in the old state, are not callable in the new state. A call sequence is only valid if it leads the FSM into a so called *final-state*. These final-states are denoted by solid black bullets within the states. In the left part of Fig. 2 only state 3 is a final-state. So, a call sequence, accepted by this FSM for example is `play-pause-play-stop`, while one cannot use a command sequence like `play-pause-stop`.

### 2.2 Architectures and interoperability checks

A basic architecture consists of components, with interoperation denoted by connections between provided and required interfaces. For instance, if `VideoStream`, `VideoPlayer` and `SoundPlayer` are components in RADL with provides-interface `provides-` and requires-interface `requires` respectively, then a basic architecture definition of TrustME is

```
architecture { VideoStream, VideoPlayer
  bind(VideoStream.requires, VideoPlayer.provides),
  bind(VideoStream.requires, SoundPlayer.provides)
}
```

The second line declares that the two components are part of the system architecture, and the third line declares that their interfaces are connected. More complex architectures can be constructed using hierarchies of components built from other components – see [17, 19] for details. A connection between two interfaces denotes a use relation, this means, call sequences of the requires-interface are sent to the provides-interface. Interoperability checks take the form of ascertaining whether or not the protocol of the provided-interface permits the calls specified by the protocol of the require-interface. Because protocols are given as FSMs, this can be computed easily and efficiently by means of the sublanguage relation between FSMs.

In this way, interoperability checks between components can be considered a form of contracts for components according to B. Meyers design-by-contract principle [12]. A contract for a component specifies, under which precondition a component $A$ has to fulfill the postcondition. Translated to interoperability checks, we check if the environment – say, a component $B$ – offers all functionality $A$ expects. We check the inclusion of the requires-interface of $A$ (precondition)

in the provides-interface of $B$ (see point (1) in Fig. 1). Is this check successful, $A$ fits into the environment and will offer all services of its provides-interface (post-condition). Hence, interoperability checks have a boolean outcome: a component fits into system or not. In contrast to other ADLs, based on more complicated behavioural semantics (such as process-calculi), our use of FSMs for protocol description means that such interoperability checks can be efficiently implemented. This is done in the *CoConut/J*-tool, enabling us to perform interoperability checks over entire architectures, by checking each connection between components. Actually, the *CoConut/J*-tool works with counter-constrained FSMs [18], an extension of FSMs which is capable of describing more complicated provides-protocols (such as the one of a stack, where no more `pop` operations must be performed than `push` operation have been performed before).

## 3 Parameterised Contracts for Protocol Adaptation

Often, a given component is not interoperable with another component. For instance, in the architecture above, `VideoStream` might generate call sequences that are not accepted by `VideoPlayer`. A common task for the software architect is to fit a component into its environment by writing adapters to wrap the component with an interface that can interoperate with other given components. In our case, because our interfaces contain protocol information, it is possible to automatically adapt components, using parameterised contracts.

While interoperability tests check the requires-interface of a component against the provides-interface of *another* component, parameterised contracts link the provides-interface of one component to the requires- interface of *the same* component (see points (2) and (3) in Fig. 1). Classical contracts do not reflect this

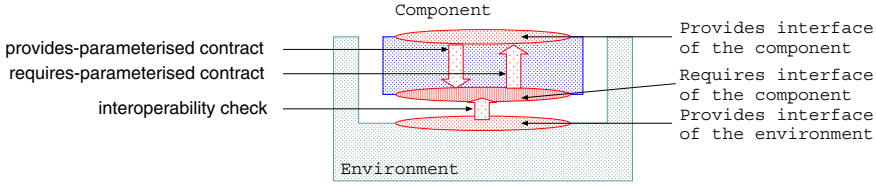


**Fig. 1.** Interoperability checks (1) and Requires-parameterised Contract (2) and Provides-parameterised Contract (3)

connection between provides- and requires-interfaces. Classical contracts, once formulated statically during component development, cannot change the post- or precondition according to a new reuse context. This motivates the formulation of two kinds of parameterised contracts: (a) provides-parameterised contracts map the provides-interface to a requires-interface. (b) requires-parameterised contracts map the requires-interface to a provides-interface.

Parameterised contracts are a mapping that is bundled with the component and computes the interfaces of the components on demand. The requires-parameterised contract takes as arguments the requires-interface of the compo-

nent and the provides-interface of the environment. Hence, parameterised contracts are isomorphic mappings between the domain of preconditions and the domain of postconditions. Both domains can be modelled as partially ordered sets (posets). (A mathematical discussion of parameterised contracts can be found in [18].) The intersection of a components requires-interface and the environment provides-interface describes the functionality which is required by the component and provided by the environment. From that information the requires-parameterised contract computes the new provides-interface of the component. Analogously, a provides-interface computes the new requires-interface from the provides interface of the component and the requires-interfaces of its clients. In the *CoConut/J*-tool, we implemented parameterised contracts for component interfaces which contain the provides-/requires-protocol. The reversible mapping between these interfaces are the so-called *service effects finite state machines* (SE-FSMs). Each service $s$ provided by a component is associated with its SE-FSM$_s$. The SE-FSM$_s$ describes all those sequences of calls to other services that can be made by the the service $s$. A simple example of a SE-FSM is shown in the example in section 4 on the right side in Fig. 2. A transition of an SE-FSM corresponds to a call of an external service. It shows that a call to service `play` of the video stream can lead to calls to the external services `play` of a `VideoPlayer` component and a `SoundPlayer` component. The latter call is not mandatory as shown by the predicate `[ifavail]`. (The context of this example is discussed in section 4.) To realise a provides-parameterised contract of a component $C$ we interpret the transition functions of the P-FSM and the SE-FSMs as graphs. Each edge (transition) in the P-FSM-graph corresponds to a service, as mentioned in section 2. We substitute for each transition in the P-FSM-graph the SE-FSM of the corresponding service. The resulting graph contains all the SE-FSMs in exactly those orders that their services can be called by the clients of the component $C$. This means, that the resulting graph, interpreted as an FSM, describes all sequences of calls to external components. Hence it describes the requires-protocol. If we ensure, that the insertion of SE-FSMs into the P-FSM is reversible (e.g., by marking the "beginning" and "end" of each SE-FSM, we can generate from a given CR-FSM a P-FSM that gives us a requires-parameterised contract. To adapt a component $C$ to an environment $E$, we build the intersection of CR-FSM$_C$ and P-FSM$_E$. The result is a possibly different CR-FSM$_C{}'$, which describes the call sequences emitted by $C$ and accepted by $E$. From that CR-FSM$_C{}'$ we generate a (possibly) new provides-protocol P-FSM$_C{}'$ using the requires-parameterised contract as sketched above. A detailed discussion of the implementation of parameterised contracts for protocol adaptation including complexity measures and advanced issues like reentrant code and recursion, can be found in [18].

## 4 Example Application

As an example we present a distributed multimedia application. From a central server, video streams are sent to various clients over network connections. Such

a video stream is not mere data, but also includes functionality such as playing, stopping, manipulation of sound and picture. An important feature is, that the clients are allowed to have different configurations regarding their hardware, operating system and applications software. A user may change these configurations frequently, e.g., by using several platforms, such as desktop computers, mobiles, hand-helds, or systems installed in automatives. These platforms differ significantly in their capabilities of reproduction, e.g., some platforms may not support sound reproduction (e.g., business desktop-systems), others cannot adjust the colours (e.g., some mobiles). Due to that variety of platforms, the server cannot simply send the video stream in thet same way to all clients nor can it assume a unique environment on all client sides for reproduction. Therefore, it is important, that the functionality provided by the video stream is adaptable to the resources available on a concrete client side.

In more technical terms, the functionality of the video stream depends on a video player and a sound player at the client side. As mentioned, the reproduction features of these tools differ significantly. The default protocol of the video stream is presented in Fig. 2 (left). This protocol represents the maximum functionality offered by the video stream. Note that the video stream offers sound and picture

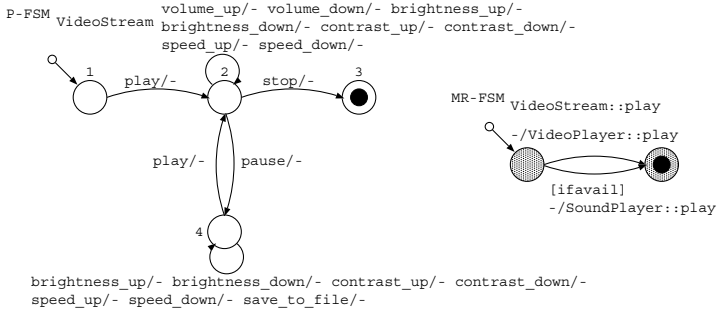


**Fig. 2.** P-FSM of the video stream (left) and SE-FSM of its service `play` (right)

manipulation while playing. However it offers only picture manipulation while pausing the video.

This protocol has to be adapted to the provides-protocols of the actual video player and actual sound player. Imagine the video stream arrives on a system without any sound support (so no sound player is given) and with a video player satisfying the provides-protocol shown in Fig. 3. Note that this `VideoPlayer` component only offers picture manipulation while playing the video.

The restriction of the video stream provides-protocol according to the concrete client environment (i.e., the `VideoPlayer`) is performed by a requires-parameterised contract. This video stream requires-parameterised contract is computed by the *CoConut/J*-toolsuite.

Fig. 4 shows the *CoConut/J*-tool with the newly computed provides-protocol. This is an adaptation of the video stream to this specific environment. Note that while pausing, it is now only possible to save to a file, but not to manipulate

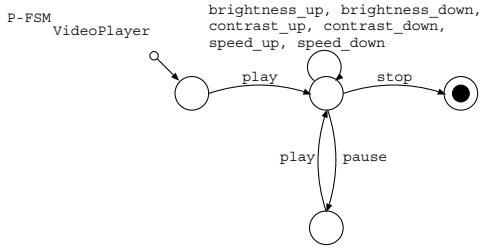


**Fig. 3.** P-FSM of the `VideoPlayer`

the picture. We should emphasise that this kind of protocol adaptation is not expressible by normal interfaces based on signature lists because the services for picture manipulation still exist in the interface. But they are not available in every state where they have been callable before.

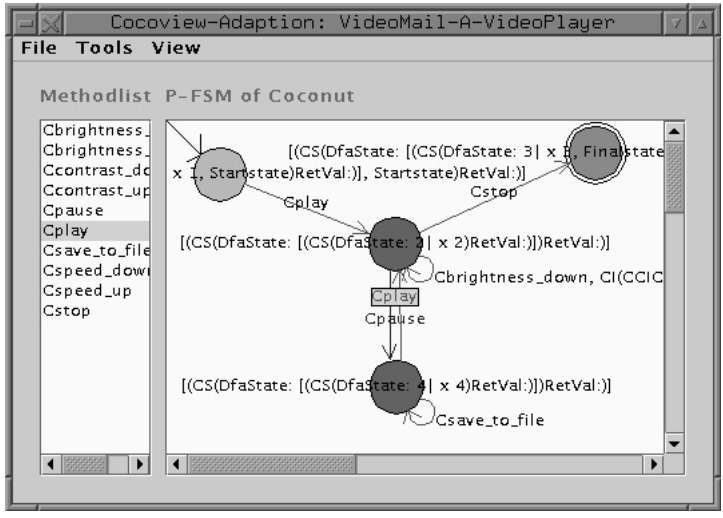


**Fig. 4.** Adapted P-FSM of the video stream as computed by the *CoConut/J*-tool

## 5 Related Work

Our focus is more specific than many ADLs, where entire component behaviour is often modelled, using stronger formalisms than finite state machines. For example, Darwin [10] uses the $\pi$-calculus [14], Rapide [9] uses partially ordered event sets and Wright [1] can use a form of CSP [6]. However, these approaches have some drawbacks when specifying protocols for architecture interoperability checks. For example, approaches using process calculi to provide more interface information are able to specify real-world protocols, but do not provide efficient algorithms for checking protocol compatibility [21]. While the properties important for architectural system configuration, such as system behaviour analysis

and substitutability checks, have sometimes been a concern during the design of component interface models, most ADLs do not explicitly focus on local interoperability checks. The use of FSMs to model protocols and test components systematically is well known from the telecommunication and distributed systems communities [7]. the use of FSMs in object interfaces was firstly described in [15]. Unfortunately, finite state machines are also among the least powerful models concerning modelling complicated protocols. Therefore, more powerful models have been derived from FSMs without loosing their beneficial properties [18]. Non-ADL based interoperability checks, using rich interface specifications, have been realised in some research tools [5, 21]. Besides our approach using interface information for automatic adaptation, [22] use interface information to create adaptors automatically. Overviews and evaluations about other adaptation mechanisms not using interface information (such as delegation, wrappers [4], superimposition [2], meta-programming (e.g., [8]), etc.) can be found in [3, 18].

## 6 Conclusion

The functionality provided by a component always depends more or less on the functionality it receives from its environment. Hence, especially in those systems, where many different configurations and environments exist, the ability to describe fine-grained changes in functionality is crucial. In our example, we demonstrated the importance of protocol adaptation, especially for loosely coupled distributed systems with frequent changes of configurations. Fine-grained changes in the availability of provided services cannot be expressed in simple signature-list based interface descriptions. They require modelling of protocol information. The underlying principle of the implementation of *CoConut/J*-tool for automatic protocol adaptation are parameterised contracts, which we presented as a generalisation of interoperability between components.

When considering an adapted component as a (higher-level) component, we are able to describe the functionality of this higher-level component in terms of its constituent inner components. Such a compositional interpretation is only possible for layered system architectures. In practice, beside layering, many other composition patterns occur. Hence, future work must be concerned with more general mechanisms to predict properties of the overall software architecture from the properties of the single components and their interaction patterns.

## References

1. Robert J. Allen. *A Formal Approach to Software Architecture.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PE, USA, May 1997.
2. Jan Bosch. Composition through superimposition. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the First International Workshop on Component-Oriented Programming (WCOP'96).* Turku Centre for Computer Science, September 1996.

3. Jan Bosch. *Design and Use of Software Architectures – Adopting and evolving a product-line approach*. Addison-Wesley, Reading, MA, USA, 2000.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.
5. Jun Han. Temporal logic based specification of component interaction protocols. In *Proccedings of the 2nd Workshop of Object Interoperability at ECOOP 2000*, Cannes, France, June 12.–16. 2000.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice/Hall, 1985.
7. Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.
8. G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4):154–154, December 1996.
9. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Apr 1995.
10. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989:137–155, 1995.
11. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Janurary 2000.
12. Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.
13. M. Migliardi and V. Sunderam. PVM emulation in the harness metacomputing system: A plug-in based approach. In J. J. Dongarra, E. Luque, and Tomas Margalef, editors, *Proc. of the 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26–29, 1999*, volume 1697 of *Lecture Notes in Computer Science*, pages 117–124. Springer-Verlag, Berlin, Germany, 1999.
14. R. Milner. The pi calculus and its applications. In Joxan Jaffar, editor, *Proc. of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, pages 3–4, Cambridge, June 15–19 1998. MIT Press, Cambridge, MA, USA.
15. Oscar Nierstrasz. Regular types for active objects. In *Proc. of the 8th ACM Conf, on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.
16. Object Management Group. The CORBA homepage. http://www.corba.org.
17. Iman Poernomo, Ralf Reussner, and Heinz Schmidt. The TrustME language site. Web site, DSTC, 2001. Available at `http://www.csse.monash.edu.au/dsse/trustme`.
18. Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.
19. Heinz Schmidt. Compatibility of interoperable objects. In *Information Systems Interoperability*, pages 143–199. Research Studies Press, Taunton, England, 1998.
20. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, Reading, MA, USA, 1998.
21. A. Vallecillo, J. Hernández, and J.M. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *ECOOP '99 Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, 1999.
22. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.