An Enhanced Model for Component Interfaces to Support Automatic and Dynamic Adaption

Ralf H. Reussner

Chair Informatics for Engineering and Science
Universität Karlsruhe
Am Fasanengarten 5, D-76128 Karlsruhe, Germany
reussner@ira.uka.de
http://liinwww.ira.uka.de/~reussner

Abstract. This paper presents a new model of software component interfaces, using an extension of finite states machines to describe (a) the protocol to use a component's offered services, and (b) the sequences of calls to the external services the component requires to fulfill its offered services. With this model we integrate information into the interface of a software component to: (a) Check whether a component will be used correctly in its environment during system integration (i.e., before the component is actually used). (b) Adapt the interface of a component which describes the component's offered services, in case the environment does not offer all resources the component requires to offer all its services. In this case the adapted component still offers a subset of its services, to the contrary of todays component systems, which do not allow any integration in this case at all.

1 Introduction

Current interfaces description languages (IDL's) model the signatures of a component's services (functions). Besides very simple components, these signatures are usually not sufficient to describe a component's applicability [VHT99]. The dependencies between service calls usually form a more or less complex protocol how to use a component. In contrast to interface information of objects or modules the interface information of components has to support component specific operations, like:

- checking, whether a component fits in an environment *while* the component is inserted in this environment, *and not later*.
- adapting a component (restricting its functionality), if a component expects more external services as the environment offers.
- controlling the extension of a component with plug-ins. A type system should provide a notion, what a component requires from a plug-in. So it can be checked in advance which plug-ins are usable.

The main contributions in this paper are: (a) An enhancement of the finite state machines model (counter automata) to circumvent the above limitations (Section 3). (b)The notion of the interface (or type) of a component, also described in 3. (c) Algorithms working on this interface information to perform automatic adaption (section 4) components. (In this paper we concentrate on component adaption, whereas

[Reu00][RH99] present mechanisms to support dynamic component extension.) A prototypic implementation is sketched in section 5.

A formal definition of counter automata and these algorithms can be found in [Reu00]. A prototypic Java implementation will be performed in the CoCoNut/J project [CoC], which enhances the existing Enterprise Java Beans component model [EJB] with our component types.

2 Related Work

Enhancing interfaces need not only be done with finite automata. Many other approaches are equally or more expressive. The notion of pre- and postconditions is used e.g., in the design by contract concept [Mey92] and in Larch [GH93]. This powerful concept could easily also be used to define call sequences implicitly. The problem of these predicate based approaches is, that checking protocol compatibility requires a model checker (or theorem prover). Other problems are not decidable. Process algebra based enhancements of interfaces [VHT99] share the benefit of a powerful and even elegant notation, but also shares the very time-complex protocol checking at composition time.

Petri nets are a very useful notation to model timing relations and synchronization of processes. Efficient algorithms for deadlock detection, live-conditions, etc. exist. So petri nets offer many benefits to enhance interfaces with protocol information (see R. van Rein in [VHT99]). We used an extension of finite automata, since we could exploit the relation between finite automata and regular languages. The adaption algorithm sketched in this paper is founded by the cross-product construction of finite automata (the intersection of regular languages). No corresponding construction of petri-nets is known, except the transformation of petri-nets in finite state machines (e.g., [Abe90]). Analogously, temporal logic also can be used to describe protocols [Hol91], but the adaption algorithm also requires a transformation into finite automata.

The deployment of a finite automata based notion to model this protocol information has been suggested by Nierstrasz (indeterministic finite automata [Nie93]) and Yellin and Strom [YS94]. The latter approach not only models the protocol to call services correctly, but also the protocol which describes the sequences to external services. This is used to generate adaptors between components [YS97]. So both protocols, the protocol defining the call sequences to offered services and the protocol describing the call sequences to external components' services are modeled in one finite automata.

On the one hand, finite automata based notations have some drawbacks. Allen describes a problem, which we will call the *or-semantic problem* (section 3.2) [All97]. The 'finiteness' of finite state machines also hinder the modeling of somehow regarded simple and elementary abstract data types like a stack, where no more `pop` operations can be called than `push` operations have been performed before. On the other hand, the use of finite automata also has some advantages, e.g., the checking whether two protocols fit can be done with a time complexity quadratic in the number of states (of the automata with the larger number of states). (Unfortunately that also can lead to a exponential complexity in the number of components to check.)

3 Our Model

3.1 Connected Protocols

Our approach defines the type (or interface) of a component as consisting of two protocols (*call interface* (a components services) and *use interface* (required external services)), in two different automata. The *Call-Automaton* (C-Aut) is an enhanced finite state acceptor, defining (at least) a subset of all allowed call sequences to a component's offered services. Figure 1 shows an example. A `VideoMail`-component offers various services, such as: `play`, `stop`, `pause`, etc. Transitions leading to an error state are omitted. The $C - Aut_{\text{VideoMail}}$ specifies for example `play pause volume_up`

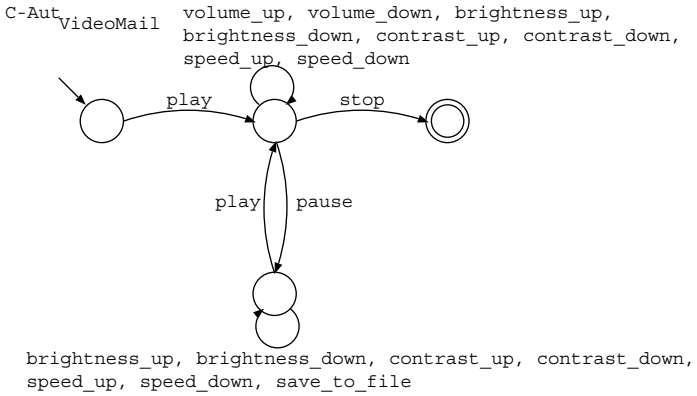

Fig. 1. Example: C-Aut of a `VideoMail` component.

`play stop` as a valid call sequence, whereas `pause stop` is an invalid sequence.

The *Function-Automata* (F-Aut's) describe for each function (service) of a component, all possible traces of calls to services of external components *for this single function*. Figure 2 shows as an example of the F-Automaton of a the above VideoMail's function `play`. Putting these protocols together we have all possible traces to external

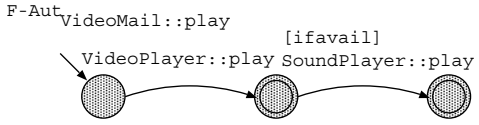

Fig. 2. F-Automaton of a Function, showing two calls to external function. First calling `VideoPlayer::play` and then `SoundPlayer::play`.

services *a component* can emit. So the Call-Automaton and all Function-Automata together form the *Enhanced Component Automaton* (EC-Aut). In pseudo-formal notion:

$$\texttt{C-Aut}_K + \{\texttt{F-Aut}_f | f \in \text{ Services}_K\} \leftrightarrows \texttt{EC-Aut}_K \tag{1}$$

Figure 3 shows a part of the EC-Aut$_{\text{VideoMail}}$ constructed according the above algorithm using the C-Aut$_{\text{VideoMail}}$ (figure 1) and the F-Automaton shown is figure 2. The advan-

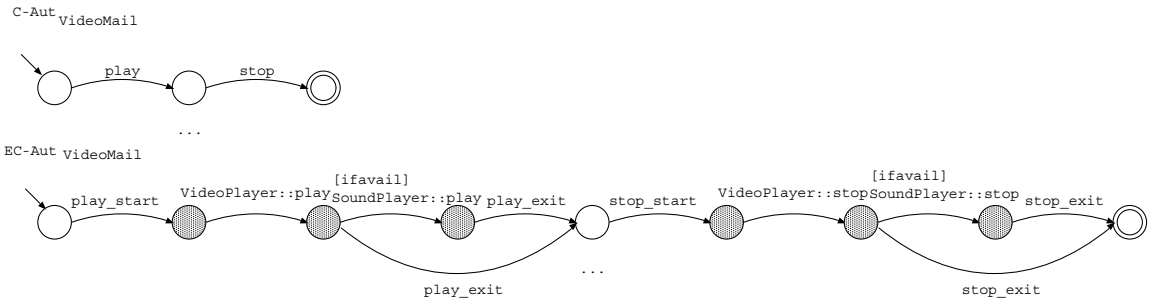

Fig. 3. EC-Aut of `VideoMail`.

tage of constructing the use protocol automaton (the EC-Aut) out of the call protocol (C-Aut), using the F-Aut's, is the linkage between call interface and the use interface. This allows more than checking whether the deployment of a component A is possible in a certain environment (i.e., checking whether all required services are available, and whether the protocols fit). Additionally, we can adapt component A, in case its use protocol does not fit exactly to the call protocols of the environmental components. To do so, we change the call protocol of component A in a way, that its use protocol matches the call protocol(s) of the environment. The procedure is explained in more detail in section 4.

3.2 Enhanced Finite Automata

As mentioned, we enhanced the model of finite state machines. Firstly, we handle that, what we call the

The Or-Semantic Problem When using a finite automata based notion to model traces, one problem occurs [All97]: what are the semantics of constructions like the one shown in figure 4(a). Function `f` may call external function `a` or external function `b`. It is clear, that one of these external functions is necessary, to bring `f` in a final state. But it is not clear, what happens to `f` in the case one of the external functions is missing. Two interpretations are possible: (1) both external function are necessary. If one is missing, function `f` cannot work. (2) if one external function is missing, `f` will not bother, since there is still one path to a final state. To decide, which interpretation is valid for this specification, more information must be supplied. We solve this, in assuming for the construction shown in 4(a) the interpretation (1) – both external functions are necessary. The other interpretation is chosen, when a condition is associated to a transition, like shown in figure 4(b). In this example, `b` is necessary, but unavailability of `a` does not hinder function `f`. Actually, this solution is simply an enhancement of the Call-Automaton's input alphabet.

Secondly, to model consequences of resource (de)allocation/creation in general, we introduced counters:

Counters One problem of the above model is, that there is no possibility to model any relations between calls. A simple but practically important relation between two calls is, that they are called equal times. Or imagine a stack. `pop` must not be called more often than `push` has been called.

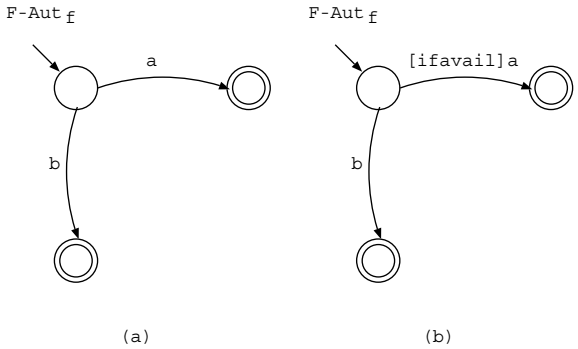

Fig. 4. Unclear semantic of construction (a). Clarified semantic of construction (b).

In general, we want to model the allocation, the release, and the creation of resources. Each service of a component may allocate, release, or create arbitrary resources. Resources can be, for example, blocks of memory, file handles, sockets, etc. We would like to check the following conditions:

1. All allocated units of a resource must be released.
2. Not more units of a resource must be allocated than created.
3. Not more units of a resource must be released than allocated. (In this case, it is all right, when a component persistently keeps some units of a resource allocated.)
4. Not more units of a resource must be allocated than released, but the release operation even works for not allocated resources.

To model these conditions we enhance the C-Automaton with counters. We allow three kinds of counters.

Error non-negative counters are integer counters which never store negative integer value. The set $C_{E-\mathbf{N}_0}$ comprises all error non-negative counters. Decreasing a counter $c \in C_{E-\mathbf{N}_0}$ which holds the value zero leads to an error-state.
Idempotent non-negative counters are integer counters which never store a negative value. The set $C_{I-\mathbf{N}_0}$ comprises all error non-negative counters. Decreasing a counter $c \in C_{I-\mathbf{N}_0}$ which holds the value zero has simply no effect.
Integer-counters: These counters represent integer values and are elements of the set $C_{\mathbf{Z}}$.

The set of counters C is defined as: $C := C_{E-\mathbf{N}_0} \cup C_{I-\mathbf{N}_0} \cup C_{\mathbf{Z}}$.

Each function of a component (== a symbol of the Call-Automaton's input alphabet)is associated with a set of counters. Those can be either increasing or decreasing. Each counter is associated with an *accepting condition*. This condition is checked, when the automaton reaches a final state. Is the accepting condition of each counter fulfilled, the automaton accepts.

We now can state our above conditions 1 – 4 in terms of our model. The condition 1, that all allocated units of a resource must be released can be modeled by a counter, which is increased for each allocation and decreased for each release. All the time this counter must be greater of equal zero. At the end this counter must be zero. The condition 2 states that never more units of a resource are allocated than are created

before. That is for a creation we increase a counter, for an allocation we decrease this counter. This counter must never be negative (but may be positive at the end). The condition 3 says, that never more units of a resource may be released than allocated. Here we increase a counter for each allocation and decrease it for each release. Again, this counter must never be negative. In condition 4, (never more units of a resource allocated than released), we increase a counter for each allocation and decrease it for each release. This counter is an idempotent non negative, i.e., a release operation can also be called, when actually nothing is to release.

Figure 5 shows the application of counters when modeling a stack, what corresponds to condition 2. In [Reu00] it is shown that the equivalence of two counter au-

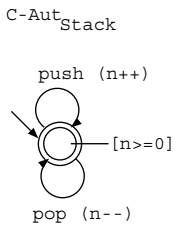

Fig. 5. C-Automaton of a stack, using counters

tomata (i.e., the type equality of two components) is decidable, when the mappings of input symbols to counter manipulations of both automata and the mappings of counters to accepting conditions for both automata are the same. (This condition is no real restriction for the practical use, since component services modifying different resources can be considered principally as different without harm.) This decidability is an important condition to compute whether a component can substitute another, or not. The algorithm to check type equivalence of two components has the time complexity of $\mathbf{O}((\max(s_1, s_2))^2)$, where s_1 is the number of states of the first automaton; s_2 of the other.

4 Component Adaption

Type adaption is a mechanism to enhance the reusability of a component through reducing its dependencies to the environment. It is based on the observation that users actually only use a subset of a component's functionality. So a restricted functionality is often sufficient for the user. More important than a full functionality is that the user has not to provide many other infra-structural resources (e.g., libraries, other components, but also system updates, etc.) which a only necessary to support the part of a component's functionality, the user actually does not need.

In terms of our types, when component A adapts to component B (the latter representing the infrastructure), then the C-Automaton of a component A restricts its functionality to its services which are supported by B. In case B supports all functionality no restriction happens. The like, when A does not need B. In praxis the most interesting and most common case is, when A needs B and C-Aut$_B$ does not offer all functionality A (in form of its EC-Aut$_A$) requires.

Our algorithm for computing the new C-Automaton of A (adapted to B), that is C-Aut$_{A\times B}$, is computed as follows:

1. Compute EC-Aut$_A$ out of C-Aut$_A$ and $\{\text{F-Aut}_f \mid f \in \text{ Services}_A\}$.
2. Compute the cross product EC-Aut$_{A\times B}$ out of EC-Aut$_A$ and C-Aut$_B$. That is the intersection $L(\text{EC-Aut}_A) \cap L(\text{C-Aut}_A)$.
3. Compute C-Aut$_{A\times B}$ out of EC-Aut$_{A\times B}$ using the $\{\text{F-Aut}_f \mid f \in \text{ Services}_A\}$.

Figure 6 shows schematically the linkage between the EC-Aut$_A$ and C-Aut$_{\text{environment}}$ and the propagation of the adapted EC-Aut$_A$ to the new C-Aut$_{A\times \text{ environment}}$. Figure 1

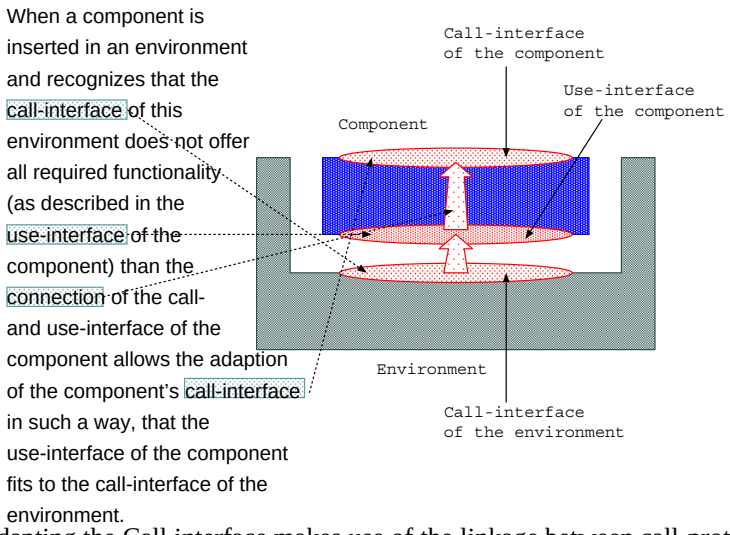

Fig. 6. Adapting the Call interface makes use of the linkage between call-protocol and use-protocol.

shows the C-Automaton of the `VideoMail` component. Consider, that `VideoMail` needs a `SoundPlayer` component and a `VideoPlayer` component. This would be expressed in its F-Automata, and hence in its EC-Automaton. They are omitted here. Imagine that the `SoundPlayer` is missing in a concrete system, because of missing hardware sound support, and that the call interface of the `VideoPlayer` is the one of figure 7. Then the new call interface of `VideoMail` coupled with `VideoPlayer` is given in figure 8. Note, that not only the sound services are not available (which would not need a protocol to model), but also the availability of the video control services (for brightness, contrast, and speed) also changes (after pause pressed). To express these changes of availability protocol information is required.

5 Technical Realization

The above described system can be implemented in several ways. One of the most important issues for the component developer is, how to state the additional type information (the C-Automaton and the F-Automata). We regard the explicit description of the C-Automaton for each component and the F-Automaton for each function as a huge

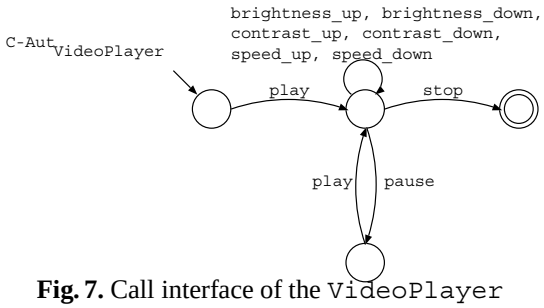

Fig. 7. Call interface of the `VideoPlayer`

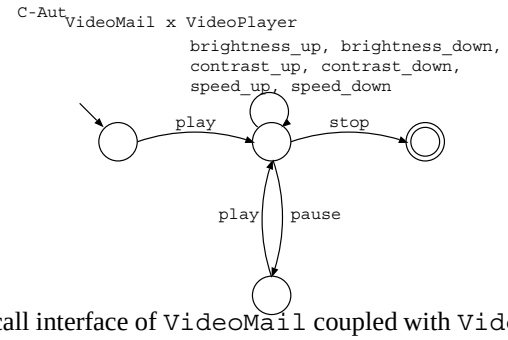

Fig. 8. New call interface of `VideoMail` coupled with `VideoPlayer`.

burden for the component developer and as practically not feasible. A practical tool will include a pre-compiler generating the F-Automata out of the control-flow of functions. The C-Automaton (defining the valid call sequences to a component's functions) can be given implicitly in simple annotations for each function, like

```
public void a()
 throws exception
 assumes b called before;
{ ...
```

Or, in a stack, making use of counters:

```
public Object pop()
 throws exception
 assumes push called at least as often;
{ ...
```

Given these annotations, a pre compiler can generate the C-Automaton.

Whether the type information is given explicitly (in a specification) or, as suggested above implicitly, the type information must be stored. Therefore, we can use either a repository or we attach the information to each object. Our prototypic implementation uses the latter approach (since not requiring a database for the repository). But the repository based approach may have benefits for large systems with an often changing configuration.

When inserting a new component A into an existing system, the run-time system checks whether the C-Automata of the component's required by A fit to A's EC-Automaton. In case of not, the runtime-system tries to restrict (adapt) A in a way, that

A calls its required components in compliance with their C-Automata. Of course, no restriction of A can make A working, when essential components are missing. (E.g., A is a statistical component that always requires essentially a mathematical package.) In this case the benefit of a type system for components is that an exception can be thrown during composition, that is *before* use. In other cases A can still offer a subset of its services, even when some required (but not essential) components are missing (E.g., A is a printer administration tool, which generally offers management of local printers and network printers. In case no network is installed, A's functionality can be restricted to only handle local printers without any consequences for the user.)

Note that anyhow, the run-time system can check statically whether the components work together (possibly after restriction of functionality), or not. This means, that during application of the components no overhead is induced. Especially, no wrapper around the components is necessary.

Our prototypic implementation heavily makes use of the Java reflection mechanism. This is one reason why we have chosen Enterprise Java Beans as the component model to enhance. In general, our type system is not bound to any specific language or component model, but some features like reflection are essential. So the integration into languages like C++ / CORBA would require is a much larger effort, since the reflection mechanism needs to be implemented first.

6 Conclusions

We presented an enhancement of finite state machines, the so called counter automata. We showed their application to extend classical interfaces to model protocol information (a) to call the offered services of a component correctly, and (b) to describe all possible sequences of calls to external services. The usefulness of this new interface information is demonstrated by the example of component adaption. Note that this adaption can be performed automatically by a component run-time system when a new component is deployed in a environment, or when the environment changes. The component developer has not to take care about possible future adaptions. The main benefit of using counter-automata as interface enhancements is the relatively low complexity of algorithms for checking the equivalence and substitutability, and for computing the adaption.

Nevertheless, this approach of modeling interfaces is still at its beginnings. Open issues are, for example:

- The hierarchical (de)composition of components: a natural property of components is, that an assembly of components can be seen itself also as a component. Conversely we should support the decomposition of a component into several other components. Possibly Statecharts[Har87] which allow a hierarchical composition of finite state machines are a good point to start research.
- Synchronization of components: Several components are often using another component simultaneously. Which type information is necessary to derive required points of synchronization?

Acknowledgements

I would like the following people for fruitful discussions and comments: Prof. Vollmar, Thomas Worsch, Kathrin Paschen, Dirk Heuzeroth, and the reviewers of this paper.

Further Information

Further information to our work can be found on the following web site:

http://liinwww.ira.uka.de/~reussner/coconuts/coconuts.html

References

[Abe90] Dirk Abel. *Petri-Netze für Ingenieure*. Springer-Verlag, 1990.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[CoC] The CoCoNut/J project homepage. http://www.ira.uka.de/~reussner/coconuts/coconuts.html.

[EJB] Sun Microsystems Corp., The Enterprise Java Beans homepage. http://java.sun.com/products/ejb/.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[Har87] D. Harel. Statecharts: a visuel approach to complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices 28(10)*, pages 1–15, October 1993.

[Reu00] Ralf H. Reussner. Formal Foundations of Dynamic Types for Software Components. Technical Report 08/2000, Department of Informatics, Universität Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany, 2000.

[RH99] Ralf H. Reussner and Dirk Heuzeroth. A Meta-Protocol and Type system for the Dynamic Coupling of Binary Components. In *Proceedings of the OOPSLA'99 Workshop on Object Oriented Reflection and Software Engineering*, November 5 1999.

[VHT99] 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.

[YS94] D. Yellin and R. Strom. Interfaces, Protocols and the Semiautomatic Construction of Software Adaptors. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 29:10 of *ACM Sigplan Notices*, pages 176–190, 1994.

[YS97] D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.