Parameterized Contracts and Adapter Synthesis

Heinz Schmidt[*]
School of Computer Science and Software Engineering
Monash University
Melbourne, Australia
hws@monash.edu.au

Ralf Reussner
Distributed Systems Technology Center
Melbourne, Australia
reussner@dstc.com

ABSTRACT
Ideal reuse takes a component as is. However, ideal reuse is a myth and hardly achieved in practice. More commonly the software architect modifies, adapts and reconfigures components. Sometimes complex synchronizations between several components are necessary before they can be deployed.

This paper presents some recent results and work in progress, on component adaptation. We develop methods for identifying incompatibilities and automatic synthesis of adapters which control components dependent on their deployment context. Typically such adapters are considerably smaller than the components themselves. They belong into the realm of connectors.

The synthesis of adapters requires compositional reasoning about extra-functional aspects of component and system behavior such as the order and timing of events, the possible matches and mismatches of such orders or partial orders. Extensions of our work currently in progress include probabilistic information associated to behavior specifications capturing usage profiles or reliability information.

Using concrete examples we show adapter generation for mismatching protocols including generating synchronizing controllers for shared resources. We only sketch our extensions to deal with reliability.

1. INTRODUCTION
Ideal reuse takes a component as is. However, ideal reuse is a myth and hardly achieved in practice. More commonly the software architect or engineer modifies, adapts and reconfigures components. Sometimes complex synchronizations between several components are necessary before they can be deployed.

[*]Currently on sabbatical at IRISA/ INRIA, University Rennes, Rennes, France

ICSE-CBSE5 '02 Orlando, Florida, USA

Industrial interface definition languages do not include sufficient information for automatic adapter or controller synthesis and component integration. Theoretical work on architecture definition and interface definition (including behavior specification) is often so general that automatic synthesis is not feasible and compatibility may even be undecidable. Our approach centers on state charts and similar representations of protocols as behavioral abstractions.

The benefit of the additional behavioral information results in a much more appropriate notion of component compatibility than offered by industrial interface definition languages. Many behavioral incompatibilities can be detected efficiently before or at deployment time rather than being left to be discovered during production runs. The provision of additional behavior contracts is a doubly-edged sword: the additional information may lead to the discovery of many more incompatibilities one might have asked for. Seemingly compatible components may be incompatible once we know precisely what is expected and guaranteed. At least for the most common and intuitively obvious adaptations, a synthesis approach is required to deal with tedious revision work resulting from enforced behavioral compatibility.

Our methods are aiming at asynchronous communicating components, i.e., the domain of open distributed processing (ODP). In such computing systems, many components exist as binaries. If we specify (symbolically) executable behaviors in our designs, we can measure, test and execute those existing binaries in partially realistic runs and with some real load data during design. This recognition is well-known and used in practice but is only slowly being absorbed in component-based software engineering research under the heading of executable analysis and verification.

The downside of ODP is that its components are often moderately large and complex servers, self-contained with their own set of resources on the one hand, yet relying on external servers on the other hand. They are therefore hard to specify and error-prone. Interoperability problems at deployment abound. Components are deployed in numerous deployment contexts. Due to the heterogeneity of these contexts (operating systems, middleware technologies and frameworks), the same ODP product exists in a myriad of slightly different configurations subject to unforeseen re-purposing. As [3] put it, adaptation is required not modification.

This paper presents recent results and work in progress on component adaptation. We study the use of communicating state charts or finite state machines as part of component behavior specification or "contracts". Our behavior contracts are parameterized by required services in the deployment context. Incompatibilities between such parameterized contracts are not only detected but we also offer semiautomatic methods for correcting them by means of adapter synthesis from the specifications of the mismatching behaviors. Typically such adapters are considerably smaller than the components themselves. They belong into the realm of connectors.

The synthesis of adapters requires compositional reasoning about extra-functional aspects of component and system behavior such as the order and timing of events, the possible matches and mismatches of such orders or partial orders. Extensions of our work currently in progress include probabilistic information associated to behavior specifications capturing usage profiles or reliability information.

Using concrete examples we show adapter generation for mismatching protocols including generating synchronizing controllers for shared resources. We only sketch our extensions to deal with reliability.

2. KENS, GATES AND PARAMETERIZED CONTRACTS

Our approach is architecture-based, i.e., it uses architectural diagrams and descriptions to provide a very high-level view of the system structure. Figure 1 shows several architectural components, some of them to be refined such as the central node *robot* with is refined in Figure 2. We call these architectural components or configurations *kens*. *Composite kens* describe configurations and protection domains hierarchically. *Primitive kens* are black-box software components, binary deployed and associated with some specification of their behavior and other relevant properties. The architectural diagrams provide very high level system views. They offer the points where we associate behavioral specifications, typically on vertices and endpoints of connections in the graph. They permit hierarchical refinement and locally confined reconfiguration while limiting changes to a subtree. Most importantly they provide the compositional structure for reasoning about behavior, and detecting or repairing incompatibility.

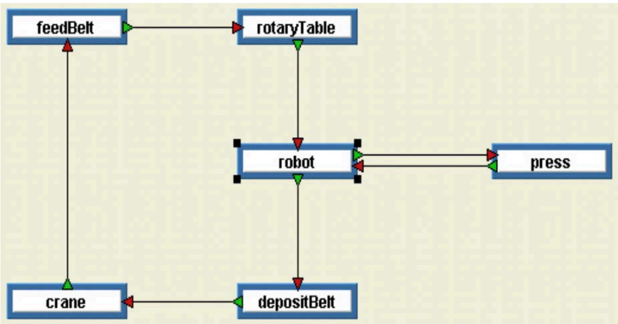

Figure 1: Component-oriented architecture diagram highlighting 'robot' in a production cell context

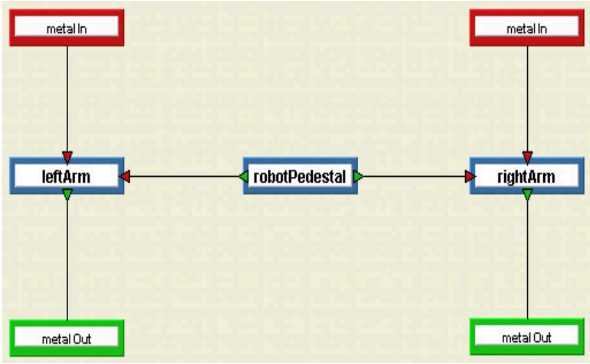

Figure 2: Hierarchical refinement of 'robot' with its two P-gates (top) and two R-gates (bottom)

Our specific architecture description derives from DARWIN and its Software Architecture Assistant which we extended in past projects to deal with inheritance, subtyping and limited contracts [10]. Our formalism distinguishes *provided* and *required* component behaviors. We associate the corresponding specifications with connection endpoints. The connections or the endpoints, if no confusion arises, are called *gates* as they serve the control and protection of access. For simplicity we refer to required gates as *R-gates* and provided gates as *P-gates*.

As kens are distributed they are not directly accessible from remote. All information or control flow, all migration of objects reaches kens through gates. Gates can potentially intercept this "traffic". Thus gates are not just interface descriptions. They are first-class objects bundling component services in meaningful ways and providing runtime adaptation and control functions according to contractual agreements following a rely-guarantee principle.

For functional units the rely-guarantee aspect of such contracts is well understood: preconditions bind the caller of the function, postconditions the supplier of the result. For components a slightly different picture emerges [8] since, for example, a required component also delivers results and hence the postconditions of its services are obligations on the environment. In a functional formalization, the component itself becomes a higher-level transformer of behavior. It takes the required behaviors and returns the provided behavior. The provided behavior $P[R]$ is thus parameterized by the required behavior R. Figure 3 sketches this dependency.

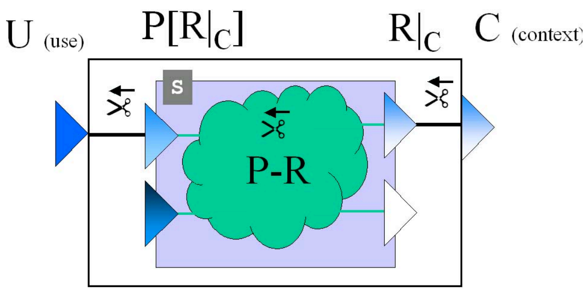

Figure 3: Deployment-Context Dependent Adaptation

dency resulting from parameterized contracts as formalized in [9]. The subsystem S is deployed in an environment of use U and service context C. Let us now assume that the actual context C does not entirely fulfill R. As the figure suggests crudely, we would then like to work backwards and compute $R|_C$, the restriction of the required behavior resulting from deployment in C. Then, using some knowledge of the transformer*P-R*, it would be desirable to compute $P[R]|_C$ (which is hopefully something like $P[R|_C]$. If the method works out, we could then extend our Architect Assistant tool to trade off the cost of extending C (to provide the required services) and that of restricting U and other affected components (iteratively rippling through the architectural context). Furthermore if the restriction operator is useful, we might be able to partly invert it and complete missing behavior. The Architect Assistant could then complement the context here and there through adapter synthesis guided by the architect.

In the reverse direction similarly we would like to be able to "cut" unnecessary behavior by some sort of reduction. This reduction would start with the context using the provided gates. It would eliminate gates that are not used and also unused services in used gates. It would utilize dependency information to compute exactly which required gates are used and which services in them. The reduction would ultimately result in (a) a simplification of the ken much like dead-code elimination in compilers, and (b) an interface reduction which can be used iteratively to reduce other (required) components in the environment - where desired. The benefit of such a reduction would not be limited to efficiency gains. More importantly it would contribute to the scalability of the architectural specification approach by keeping the composite specifications as reduced as possible.

We have developed the necessary formalization and a prototype tool to do exactly that. We are currently extending it and have started the integration with other tools. In our

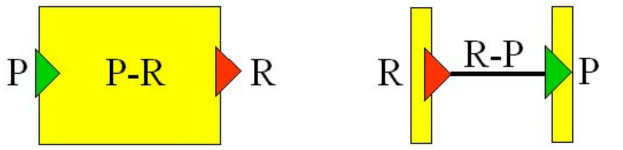

Figure 4: Dependencies result from behavior transformations and bindings

project we have studied suitable finite state machine (FSM) and Petri net representations. In this paper we only sketch the approach and highlight some problems, and also restrict this for simplicity to FSMs. More details can be found in [8, 11].

Figure 4 shows two types of dependencies we need to consider for chaining together reasoning steps through architectures. *P-R* represents those dependencies between P and R leading to parameterized contracts. *R-P* represents dependencies resulting from bindings of systems in a deployment context. In our semantics, *P-R* dependencies are resulting from the construction of black-box components or from their architectural composition into larger-scale assemblies while *R-P* dependencies are the result of gluing and possibly of adaptations. In particular some of the adapters will be multi-connectors, fanning in and out to multiple kens.

As we are interested specifically in behavioral aspects and extra-functional properties resulting from behavior, we specify the abstract types of messages and FSM descriptions characterizing the protocol for calls passing through the gate, i.e. order constraints on those calls.

Now the *P-R* dependencies of primitive kens are explicitly specified by so-called *service-effect protocols*. Such protocols associate a separate FSM with each service name in each provided (formal) gate. We use these to compute an abstract transformer behavior for the ken. The service-effect protocols of composite gates are recursively and compositionally computed from it subkens and their actual bindings.

3. ADAPTATION PROBLEMS

The following fragments of a multi-media mail system (including video, audio and text) are to illustrate specific adaptation problems. These arise when R-P bindings do not match, when they are connected to multiple gates or kens. Generally these problems occur in the composition of architectural configurations and assemblies, i.e. composite kens. Removing these incompatibilities leads to a kind of normalized architecture where the multiple P-gates or R-gates of a ken are properly synchronized into a single virtual P- or R-gate respectively and where gates are 1:n or n:1.

3.1 Problem 1: Split Adapter

Given n P-gates and 1 R-gate, construct a minimal adapter dispatching the calls appropriately to the shared R-gate according to its protocol (and assuming conflicts are excluded, i.e., calls to the same service).

For example in Fig. 5 the `VideoPlayer` and `SoundPlayer` protocols are provided and synthesized into a single P-gate to serve a single required behavior for multi-media mail. The construction is based on shuffle automata[11]. The shuffle protocol $A \oplus B$ we use here has as states the Cartesian product $A \times B$ of the states of the input FSMs, as input alphabet the union $I_A \cup I_B$ of the respective inputs (which by assumption is disjoint). Its transitions $t((a,b),i)$ move from state (a,b) to $(t_A(a,i),b)$ or $(a,t_B(b,i))$ depending on whether $i \in I_A$ or $i \in I_B$.

The complexity of this algorithm lies mainly in the definition of transitions. The number of resulting transitions is never larger than $|S_A| \cdot |S_B| \cdot (|I_A| + |I_B|)$. The algorithm produces an `1:2`-adapter, which can be applied associatively multiple times to create an `1:n`-adapter.

3.2 Problem 2: Join Adapter

As a motivation for the next problem consider a consumer and producer accessing a shared buffer (Figure 6). The producer writes to the buffer; the consumer reads from it. The producer and consumer have been integrated into a larger system and require synchronization. If the shared resource provides sufficient synchronization, we should be able to "join" the required protocols and check the result against the single provided protocol of the resource. Of course it is not sufficient to check each of the required gates against

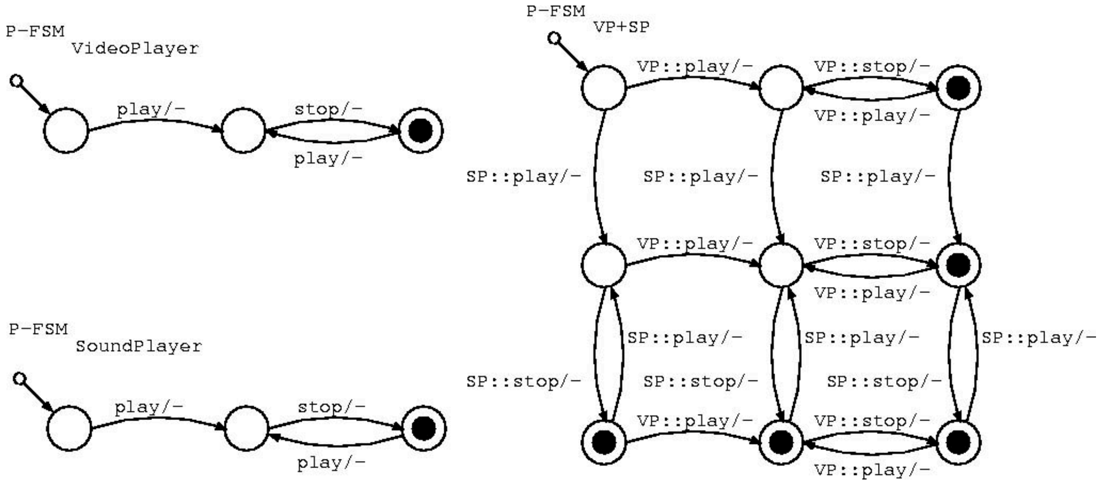

Figure 5: n:1 Adapter

the resource because this would ignore the required synchronization. For simplicity our figure shows the protocols for a buffer of size 1. This means, producer and consumer communicate using a simple handshake protocol.

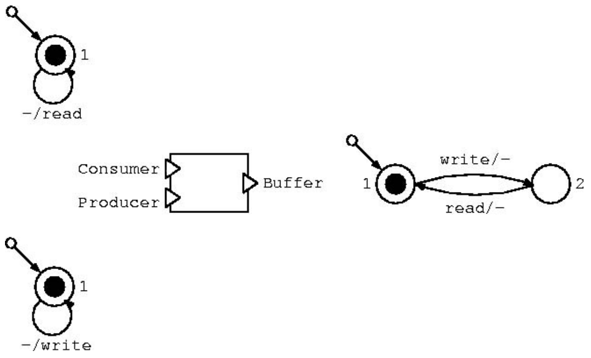

Figure 6: Synchronizing Adapter

Abstractly we formulate the problem as follows: *Given n R-gates and 1 P-gate, combine the behaviors into a single maximal behavior synchronizing the calls according to the shared provided resource protocol of the P-gate.*

The solution to this problem is more involved using the shuffle $A + B$, this time with possibly intersecting input alphabets. This means, we move like A if the input is solely in A, like B if the input is solely in B and simultaneously if it is in the intersection. Then we construct the intersection FSM $(A+B) \times C$ with the provided (context) gate FSM C. This intermediate behavior is used to compute the synchronization as the solution of a data flow problem considering adjacent edges (transitions) e, e' whose owner (the FSM A or B) switches. In other words communications from A to B or B to A are necessary to synchronize those transitions. We simply signal the corresponding input in e and wait for the signal in e'. Our tool constructs the annotations in UML state chart notation. The actual construction is a little more involved than sketched here [11].

3.3 Problem 3: Smart Injection Adapter

Incompatibility is the companion of reuse (attempts). Especially the more we formalize behavior, the more incompatibilities come to the fore. Our case studies show that many of these incompatibilities can be overcome by "dog-work" – many little tedious and boring revision steps. Why not synthesize an adapter? The adaptations do not have to occur entirely behind the back of the software engineer. Through interactive dialog the engineer may accept, reject or guide the process.

Fig. 7 shows two CD player protocols that are incompatible. Compatibility corresponds to a subtype relationship between behaviors. An FSM is a subtype of another, if and only if its language is a superset of that of the other FSM. In other words, it accepts the sequences generated by the other FSM. Here the incompatibility is due to the provision of a richer interface than required but unfortunately in such a way that the additional feature require additional actions.

Abstractly we formulate this problem as that of *inserting* call sequences when we get "stuck" in a state. The desired next state may be known, or the desired next service call. However the provided service may require the call sequence before allowing the desired invocation. Similarly the caller may reach a final state without properly terminating the resource. In each case, the extra call sequence can be computed. Yet another shuffle synthesizes the minimal adapter to solve this problem, if there is a unique solution. If the choice is ambiguous the architect must guide the construction process or at least the construction tactics.

4. PARAMETERIZED RELIABLE PROTOCOLS

We have started extending the ideas of parameterized contracts and adapters to Markov Chains based on work by Wang and others[13]. Their work associates single reliability with a component and uses the graph of component composition for composite reliability calculations. This is appropriate for simple dependencies such as signal flows or streams typical for hardware components. For complex ODP software components there are, however, many different call pat-

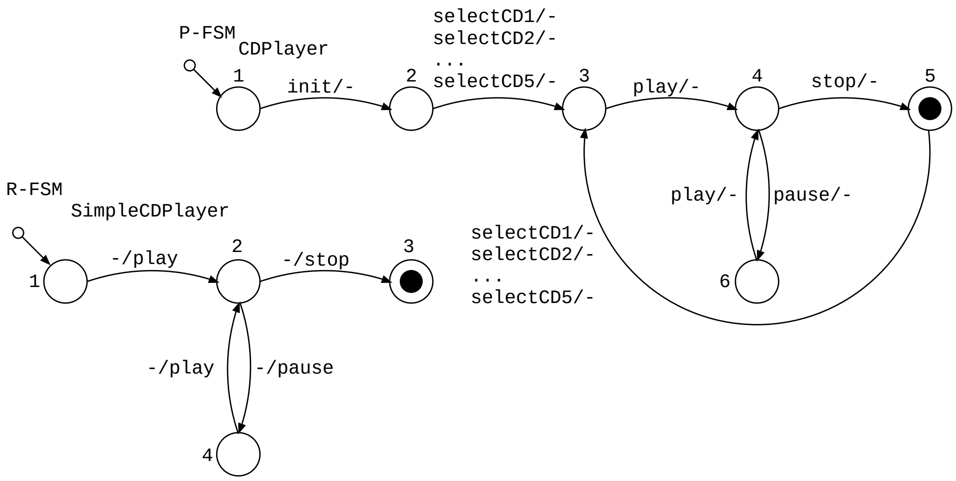

Figure 7: Incompatible CD player protocols. Normally incompatibility is a reuse stopper. Here a smart adapter injects the necessary initialization and CD selection sequences.

terns to required components and their services. Consider for example two required gates *G1* and *G2* invoked by a composite ken K. If *G1* provides access to services (of some other ken) of reliability 1.0 while *G2* provides access to services of reliability 0.5, it will depend on the usage profile of these services how the given reliabilities have to be combined to provide an accurate reflection of the composite reliability of K. The composite reliability is not only parameterized by the deployment context (here $R_{G1} = 1.0$ and $R_{G2} = 0.5$. It is will also depend on (a) the usage profile of the services K provides and (b) the service effect protocols, i.e., architectural decisions determining the *P-R* dependencies between those provided and required services. For example, assume that one of the provided services of K, $p1$ is called two orders of magnitude more often than another service $p2$ in the given deployment context. Furthermore assume that $p1$ invokes a loop repeatedly requesting services from *G1*, while $p2$ invokes *G2* exactly once every time it is called. Then the reliability of *G2* will become dominant in the composite reliability R_K.

The parameterized contracts sketched in this paper allow us to perform such calculations accurately. First case studies determining the accuracy of our calculations are currently being reviewed for publication.

In our reliability extension we assign probabilities to some transitions and compute others. Firstly, provided services of composite kens indicate their usage profiles by means of probability distributions, i.e., the likelihood of invocation in the deployment context. Probability distributions for hidden transitions in that service effect protocols of primitive kens (black-box components) capture design decisions regarding their internal resources without revealing implementation details. Reliability is captured for primitive kens only and for bindings reflecting network availability data. With suitable methods, we now compute the missing probabilities so that all FSM provided protocol services have associated usage profiles and all FSM required services have associated reliability predictions. This is possible using the *P-R* protocol compositions for composite kens.

FSMs with transition probabilities are Markov chains here parameterized by a set of probabilities associated to the deployment context. The result of our reliability calculations (not the calculations themselves) can be interpreted as follows. Intuitively every possible call chain through several components (primitive kens) and through the connecting middleware (composite kens, adapters and connectors) defines a possibly long product of reliabilities. Each such product defines the likelihood of the success of the entire chain, given any of the services in the chain can fail. All possible such chains (infinitely many and of increasing length) are summed up to define the composite reliability. The theory of Markov chains allows solving such problems analytically as the limits of certain geometric series.

Our parameterized protocols, especially the *P-R* service-effect protocols and the *R-P* adapters provide the link to bring together the usage profile and reliability in the deployment context in order to reason about these extra-functional behavioral properties in a context-dependent but compositional way. Ultimately this means, those extra-functional properties are specified as a function of properties of the deployment context. The contracts between components and environment are thus parameterized contracts that are actualized at integration or, if appropriate, deployment time.

5. RELATED WORK AND PROBLEMS

Architecture definitions take a mix of black-box and gray-box approach revealing configuration details in terms of succinct interface definitions. This approach is taken in OLAN [1], for instance, or in DARWIN [6]. and in our own DARWIN extension [10, 5]. A general overview over ADLs is given in [4].

State machine based approaches are abundant in ODP. They have been used for protocol and interface descriptions [7, 12, 2]. However adapter synthesis is relatively new and existing work has focused so far on either sequential protocols or the specifications of single machines without distinguishing provided and required interfaces. The resulting synchronization restricts connectors to simple signal flow often unbuffered. In our approach connections themselves exhibit complex behavior and can serve modeling complex message flows between large-scale components.

Current industrial component models, such as Microsoft's COM+ or .NET and Sun Microsystems' and IBM's EJB model the interface of a component (object) as a list of their services without regard for call orders or synchronization. Moreover without the distinction of required and provided services dependencies cannot be modeled.

Our approach has opened a number of problems. So far, adapter synthesis has developed bottom up as a result of case studies and the study of conformance and compatibility issues in the context of parameterized contracts. An adapter calculus is missing including common architectural composition and adaptation operators.

For the reliability calculations we have yet to analyzes carefully the impact of adapter generation on reliabilities. Intuitively the shuffle constructions preserve reliability as the order of factors in products does not matter.

In a parallel project we study state machine decomposable Petri nets, to model true concurrency of communicating state machines and bounded counter FSMs. Counters are needed in various synchronization disciplines and have a natural representation in Petri nets as bounded places, which accept a bounded number of tokens. Unfortunately, arbitrary counter machines have Turing power, their intersection is non-decidable and thus they do not render themselves to the constructive and decidable approach presented here.

6. CONCLUSIONS

This paper sketched an FSM-based specification of architectural elements in component-oriented systems. FSMs capture protocols enriching kens (architectural units) and gates (connectors and adapters) with abstract behavior and dependency specifications. We motivated the notion of parameterized behavioral contracts, which determine critical component properties only at system integration or deployment time.

Parameterized contracts were then used for compatibility analysis and adapter synthesis using small examples. We sketched the construction of some adapters highlighting their ability to overcome incompatibility problems by means of compositional reasoning about, and prediction of, extra-functional properties such as synchronization, timing or reliability associated to component behavior.

Finally we summarized some related work and open problems.

7. REFERENCES

[1] R. Balter, L. Bellisard, F. Boyer, M. Riveill, and J.-Y. Vian-Dury. Architecturing and configuring distributed applications with olan. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 15–18, 1998.

[2] L. de Alfaro and T. A. Henzinger. Interface automata. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, Sept. 10–14 2001. ACM Press.

[3] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, Reading, MA, USA, 1999.

[4] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley, Reading, MA, USA, 2000.

[5] S. Ling, H. Schmidt, and R. Fletcher. Constructing interoperable components in distributed systems. In *IEEE Proceedings of TOOLS Pacific '99, Melbourne*, pages 274–284. IEEE Computer Society Press, 1999.

[6] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of ESEC '95 - 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153, Sitges, Spain, 25–28 Sept. 1995. Springer-Verlag, Berlin, Germany.

[7] O. Nierstrasz. Regular types for active objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, Oct. 1993.

[8] I. H. Poernomo, R. H. Reussner, and H. W. Schmidt. Architectures of Enterprise Systems: Modelling Transactional Contexts. In *accepted for the Proceedings of the First IFIP/ACM Working Conference on Component Deployment*, June 2002.

[9] R. H. Reussner. The use of parameterised contracts for architecting systems with software components. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001.

[10] H. W. Schmidt. Compatibility of interoperable objects. In B. Krämer, M. P. Papazoglou, and H. W. Schnmidt, editors, *Information Systems Interoperability*, pages 143–181. Research Studies Press, Taunton, England, 1998.

[11] H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronisation. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS) V*, pages 213–229. Kluwer Academic Publ., March 2002.

[12] A. Vallecillo, J. Hernández, and J. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *ECOOP '99 Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, 1999.

[13] W.-L. Wang, Y. Wu, and M.-H. Chen. An Architecture-Based Software Reliability Model. In *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing, 16-17 December, Hong Kong, China.* IEEE, 1999.