Systematic aid for developing middleware architectures

Valérie Issarny, Christos Kloukinas, Apostolos Zarras

Systematic Aid in the Development of Middleware Architectures

Valérie Issarny Christos Kloukinas Apostolos Zarras

INRIA
Domaine de Voluceau
Rocquencourt - Le Chesnay - 78153
France

{Valerie.Issarny,Christos.Kloukinas,Apostolos.Zarras}@inria.fr

To appear in Communications of the ACM vol. 45 no. 6

The use of middleware is the current practice for developing distributed systems. Developers compose reusable services provided by standard middleware infrastructures, *e.g.*, CORBA [1], DCOM[2], Java RMI & related services[3], *etc.*, to deal with problems like distribution, security, transactional processing, fault tolerance, *etc.*. The development process gets even easier after the evolution of the originally proposed Object-Oriented middleware paradigm, towards nowadays component-based middleware paradigm, *e.g.*, CCM[1], MTS[2], EJB[3]. Developers do not have to burden with the, sometimes considerably complex, composition of different middleware services. Instead they build their middleware components and deploy them within off-the-shelf middleware containers realizing a customizable composition of middleware services.

However, things are not as simple as they seem. Middleware vendors have to design and implement complex architectures combining available middleware services into a flexible/customizable way. Different ways of composing middleware services into a middleware architecture that satisfies application requirements are possible. The resulting compositions should be supported by the configurable middleware architecture that is provided by vendors to customers. Moreover, off-the-shelf middleware architectures should come along with a quality assessment of the different possible compositions they support. This shall give customers clues for selecting the most suitable middleware compositions for their particular systems.

In this paper we address the above issues. More specifically, we present a developer-oriented environment that facilitates the design and quality analysis of flexible/configurable middleware architectures. The environment provides support for modeling middleware architectures. It further comprises a repository that is populated by a middleware vendor with basic middleware architectural patterns describing the use of individual services provided by the vendor's infrastructure. The environment further provides a tool which constructs all possible valid compositions of a set of basic middleware architectural patterns. Finally, the environment includes tool support for the automated generation of traditional quality models for the performance and reliability analysis of the different valid compositions of middleware architectural patterns. Those models serve as input to existing performance and reliability analysis tools, which are integrated into the environment.

1 MODELING MIDDLEWARE ARCHITECTURES

To model middleware architectures we use an Architecture Description Language *(ADL)* proposed in [12]. As most typical ADLs, the one proposed in [12] provides basic modeling constructs for the specification of:

[1] http://www.omg.org/technology/documents/spec_catalog.htm\#CORBAspecs
[2] http://www.microsoft.com/com/wpaper/compsvcs.asp
[3] http://java.sun.com/products

(1) components, *i.e.*, units of data, or computation; (2) connectors, *i.e.*, the interaction protocols among components; (3) configurations, *i.e.*, the assembly of components and connectors.

For the specific case of modeling middleware architectures, our ADL provides subtypes of the basic architectural elements representing middleware specific architectural abstractions like stubs, interceptors, containers, message oriented connectors, stream connectors, RPC connectors, *etc.*. The definition of those abstractions are inspired by well-known existing middleware standards, *e.g.*, CORBA, RM-ODP[4] etc.

To increase the impact of our environment in the real world we further defined the relations among the basic constructs of our ADL and standard UML [5] elements. Based on the previous we defined ADL components, connectors and configurations as UML stereotypes extending the semantics of standard UML elements like subsystems and associations.

Since the basic ADL constructs are extensions of standard UML elements, we can use any existing UML modeling tool for the specification middleware architectures. Rational Rose [6] is a well-known such tool, which is integrated into our environment. Rose allows the definition of user specific add-ins that facilitate the definition and use of stereotyped elements. Using the aforementioned facility, we implemented an add-in that eases the specification of middleware architectural descriptions.

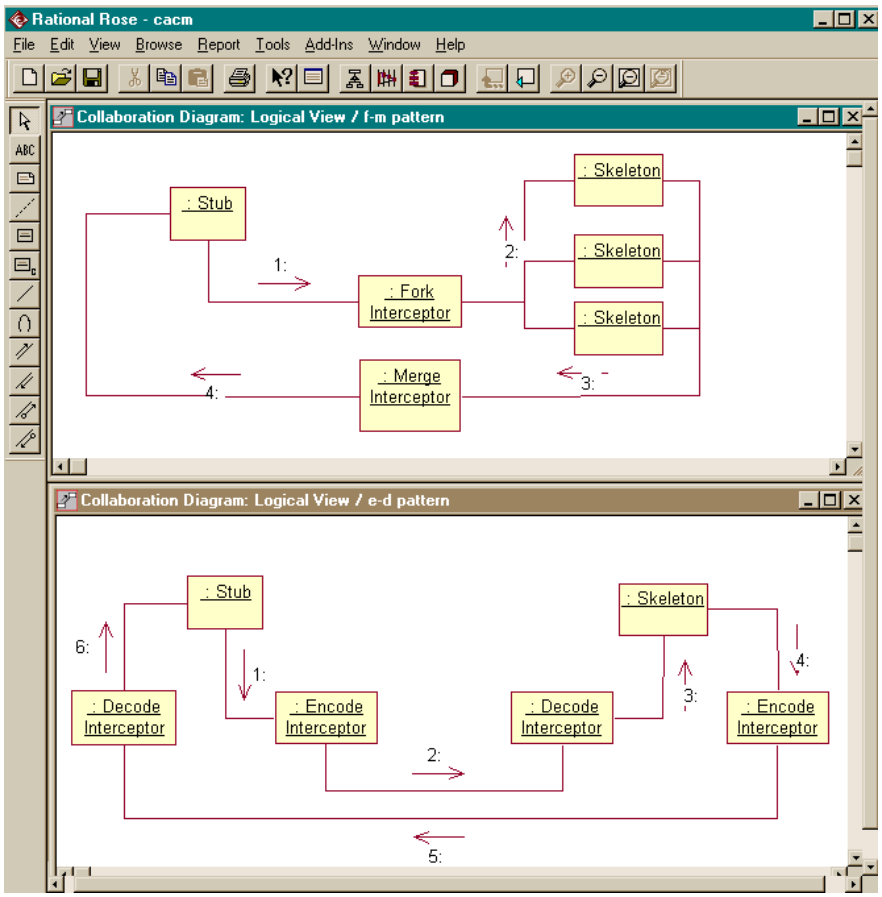

Figure 1: Architectural patterns for fault tolerance and security

Our environment further includes a repository populated with the architectural specifications of existing middleware infrastructures resulting from standards and documentation provided by the vendor of the infras-

[4] http://www.iso.ch:8000/RM-ODP

[5] http://www.omg.org/technology/documents/spec_catalog.htm\#modelingspecs

[6] http://www.rational.com/products/softdev.jsp

tructure. The architectural specification includes at least the definitions of the basic architectural elements of the infrastructure, *e.g.*, CORBA ORB and services, and architectural patterns describing/constraining the correct use of the basic architectural elements. Figure 1 gives two simple examples of such patterns describing/constraining respectively the use of the CORBA security and fault tolerance services.

According to the CORBA security service standard specification [7] client requests and server replies pass from two layers of interceptors. The first layer performs access control and auditing, while the second preserves integrity and confidentiality via the use of existing cryptographic mechanisms. For the sake of simplicity, in the pattern given in the upper collaboration diagram depicted in Figure 1 we take into account only the second layer of interceptors.

According to the standard for fault tolerant CORBA [7] a CORBA compliant infrastructure should support both the passive and the active replication styles. In the former style a client communicates with a replicated group of servers using simple IIOP invocations to the primary member of the group. In the latter case, a client communicates with the group through the use of a proprietary multi-cast group communication protocol. The lower collaboration diagram in Figure 1 describes this pattern. In particular, the ForkInterceptor component replicates client requests, while the MergeInterceptor makes sure that only one reply is delivered to the client. Total order of request delivery is preserved by the functionality included in the ORB connector.

The individual patterns describing/constraining the use of the security service and fault tolerance are the basic input allowing us to generate all possible valid compositions of them towards building a secure and fault tolerant middleware architecture.

2 COMPOSING MIDDLEWARE ARCHITECTURES

2.1 Issues in composing software architectures

So far there has been a number of different approaches related to the composition of software architectures. In [8] the authors propose to compose two architectures by merging components that are present in both of them. When the initial architectures do not share any, they propose introducing a new "bridge" architecture containing a component from one architecture and another component from the other, so that the two systems can communicate. Even though this is an interesting approach for constructing a system by composing other systems, it is not so useful for composing middleware architectures. Applying the approach to the example of the two architectural patterns we gave in Figure 1, while assuming that the different interceptors do not share any features, gives us a middleware architecture with a client stub, replicated skeleton and two different connection paths between them. One of them enables secure communication while the other realizes the multi-cast communication protocol. Naturally, what we would need here is a single connection path build out of the middleware components in Figure 1, which provides a secure multi-cast communication protocol.

Another interesting approach on composition is proposed in [1].The authors present a method for synthesizing linear architectures, i.e., systems where each component has a single input and a single output port. To synthesize a system, they ask the user to provide them with a linear time temporal logic property, which constraints the structure of the resulting composite architecture. Then, they synthesize all possible compositions of the available components that match this property. So, unlike [8] where the authors try to find the "best" composition, the authors of [1] try to construct all possible compositions, which is our goal as well. However, unlike [1], we do not want to restrain the architectures to be linear ones, as it is not always the case for middleware architectures (the architectural pattern for security in Figure 1 is linear but the one for fault tolerance is not). Instead we wish to obtain a method which can work with any kind of architectures.

Another approach to composition of middleware architectures can be found in [10], where the authors introduce a set of operators which transform generic communication mechanisms, e.g. RPC, to incrementally add new capabilities/non-functional properties. In their paper, they give an example of transforming the Java Remote Invocation mechanism to one that supports Kerberos authentication. However, there is a basic problem with this approach, at least with respect to our goal: the transformations they consider are build

[7] `http://www.omg.org/technology/documents/spec_catalog.htm\#CORBAservices`

manually by the architect and their application results in a single composite connector. Instead, we would like to automatically and without having to describe possible transformation operators, obtain all possible compositions. This would not only allow the architect to choose a composite connector according to the characteristics of the system he is trying to describe, but to explore new and unexpected ways of using middleware components as well.

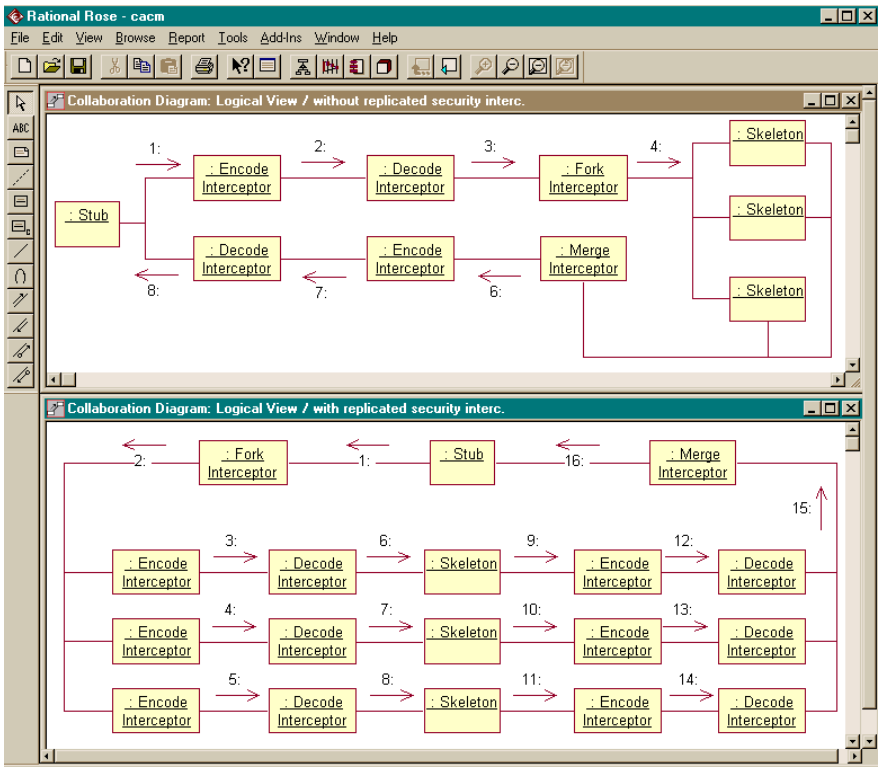

Figure 2: Two valid compositions of the architectural patterns for fault tolerance and security

2.2 Automating the composition of middleware architectures

To summarize our goals, we aim at providing an automatic method for constructing all possible compositions of two middleware architectures that satisfy the application requirements, *e.g.*, for security and fault tolerance. To do so, we have to examine all possible interconnections of the middleware components, to find the ones that provide the required properties. Since middleware components are supposed to be highly reusable, we can consider interconnections More specifically, we assume that one middleware architecture is the basic one and then interpose the architectural elements of the other in it.

We have investigated two solutions, to achieve the previous. In the first [6], we use a model checking tool to connect middleware architectural elements in a non-deterministic manner. Moreover, we specify application requirements, *e.g.*, security and fault tolerance, as constraints that should not hold for compositions. Given the previous the model checker provides us with all counter examples for which the constraints hold, *i.e.*, compositions that satisfy application requirements. However, the search space is usually be too large to search exhaustively.

Therefore, we have chosen a second solution, which is based on the structural information present in the initial architectures. The idea behind it is based on the fact that in most, if not in all, valid compositions, the initial data-flows will be preserved. As a consequence, we can construct only those compositions which preserve the initial data-flows. If we constrain ourselves to linear architectures, *i.e.*, strings of let-

ters $A =$ "$\alpha_1 \cdots \alpha_n$" and $B =$ "$\beta_1 \cdots \beta_m$", with k and l respectively the number of application component representatives, *i.e.*, stub/skeleton components in case of CORBA, in each one, then the structurally valid compositions are constructed as follows. We construct strings of length $n + m - l$, since the roles of the l application components of B will be played by components of A. In these, we identify n places at which we can place the components of A, which means that we obtain at most $\binom{n+m-l}{n}$, since the order, *i.e.*, data-flows, of letters, *i.e.*, components, must be preserved. In reality, there will be a lot fewer compositions, because we use additional structural constraints, see [5].

So, if we try to compose the security and fault tolerance architectural patterns given earlier, then we obtain only 23 structurally valid compositions, two of which are given in Figure 2, while one would expect about $\binom{6+6-2}{6} = 210$ compositions. Middleware architects can select among these solutions by verifying which of them provide a particular temporal logic property, *i.e.*, by model checking them, by doing performance, reliability analysis, *etc.*.

3 ANALYZING THE QUALITY OF MIDDLEWARE ARCHITECTURES

3.1 Issues in the quality analysis of software architectures

Pioneer work related to the quality analysis of systems at the architectural level includes Attribute-Based Architectural Styles *(ABAS)* [4] and the Architecture Tradeoff Analysis Method *(ATAM)* [3].In general, an architectural style includes the specification of types of basic architectural elements, *e.g.*, pipe and filter, that can be used for specifying a software architecture and constraints on the use of those elements. An ABAS additionally includes support for modeling and analyzing the quality of the architecture regarding a particular quality attribute, *e.g.*, performance, reliability, availability, *etc.*. ATAM is a method for using ABAS in conjunction with a set of possible use case scenarios towards performing tradeoff quality analysis. ATAM has been used for quality analysis regarding attributes like performance, availability, modifiability, and real-time. In all those cases, quality attribute models, *e.g.*, Markov models, queuing networks, *etc.*, are manually built given the specification of a set of scenarios and the ABAS-based architectural description. However, in [3], the authors recognize the complexity of the aforementioned task. As stated the development of quality analysis models requires about 25% of the time spent for applying the whole method. ATAM is a promising approach for doing things right. Nowadays, however, there is a constant additional requirement for doing things fast and easy. The previous is emphasized even more in the case of middleware development since middleware architects, designers and developers are quite familiar with certain industrial standard development methods, *e.g.*, UML and RUP [8], but have no experience in using formal analysis methods.

To deal with the previous issues, the environment we propose includes automated procedures for the generation of quality analysis models from middleware architectural descriptions. In particular, we support the generation of queuing networks for performance analysis and state space models for reliability/availability.

3.2 Automating the quality analysis of middleware architectures

In general, building model generation procedures involves defining the relationships among the basic concepts used for modeling middleware architectures and the basic concepts assumed by the formalism used for the quality analysis. Moreover, the generation procedures should be customizable regarding the features of particular middleware infrastructures offering the basic services used for building the middleware architecture that is the subject of the analysis (see [11] for the specific case of CORBA Workflow-Based middleware infrastructures).

Taking, for instance, the case of reliability the basic measure we assume is the probability that a collaboration describing/constraining the use of a middleware architecture successfully completes within a given duration representing the lifetime of the application using the middleware architecture. A collaboration may

[8] `http://www.rational.com/products/proman.jsp`

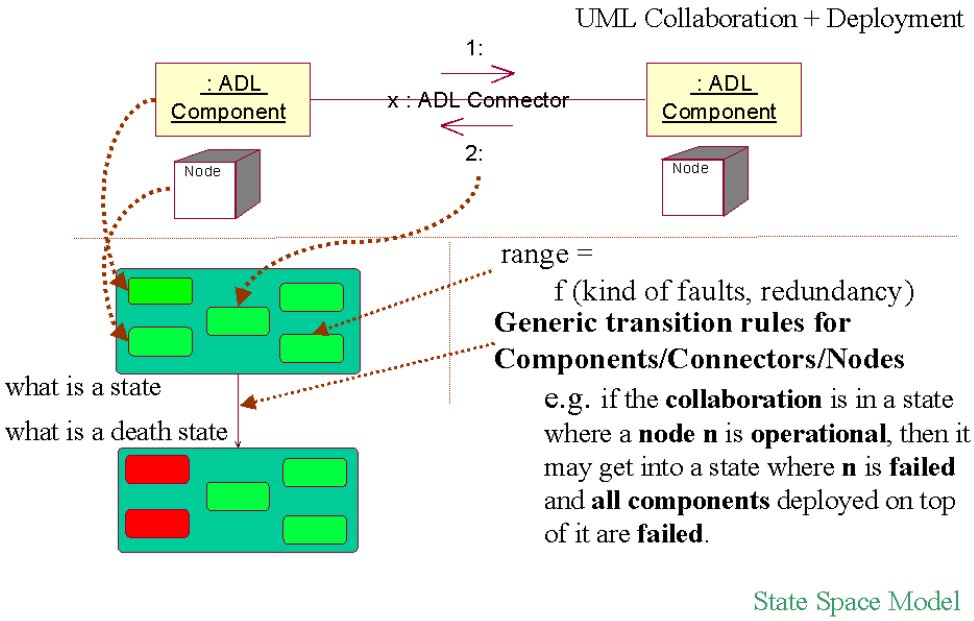

Figure 3: Mapping between ADL elements and traditional formalisms for quality analysis.

fail if instances of middleware components, connectors, nodes, used, fail because of faults causing errors in their state. The manifestations of errors are failures. Faults and failures can be further characterized by properties like the persistence and the arrival rate of faults, the domain and the perception of failures, *etc.* [7]. The range of values of those properties are specific to the middleware infrastructure used and can be obtained using measurement based techniques (*e.g.*, [9]). Different combinations of those values are used to customize properly the model generation procedure.

Except for faults and failures, another important parameter affecting reliability is replication. A replication group is a configuration of redundant components, which behave as a single fault tolerant unit. The replication group is further characterized by the kind of middleware mechanisms used to detect errors (*e.g.*, voting, acceptance tests), the way the constituent elements execute towards serving incoming requests (*e.g.*, in parallel, sequentially), the number of component and node faults that can be tolerated, *etc.*. The range of the values of the previous properties depends on the different policies provided by the particular middleware infrastructure.

Having the description of a middleware architecture, which includes the specification of the above properties we can generate a state space model for reliability analysis. The resulting state space model shall consist of a set of transitions between states of the collaboration. A state describes a situation where either the collaboration may successfully take place, or not. In the latter case the state is called a death state. Figure 3, gives more details regarding the generation procedure. More specifically, parsing the collaborations allows generating:

1. *A definition of what is the state of the collaboration*; The state of a collaboration is composed of the states of the middleware component and connector instances used within the collaboration and the state of nodes on top of which the component instances execute. The range of states for a component/connector/node depends on the kind of faults that may cause failures. At this point, the generation procedure must be customized accordingly.

Cases	#transitions	reliability (upper bound)	reliability (lower bound)
composition B. interc.	12	0.70	0.67
composition A - single version security service	24	0.74	0.72
composition A - multi version security service	48	0.80	0.79

Table 1: Results from the reliability analysis

2. *A definition of what is a death state*; The death state of a collaboration is a state where any of the instances, or nodes used within it is not operational. Hence, the death state definition is the disjunction of base predicates, each one of which defines the death state constraint for an individual instance used in the collaboration.

3. *A set of rules describing transitions among sets of states of the system*; Transition rules are generated based on the application of generic transition rules pre-defined for the different kinds of architectural elements, *i.e.*, stubs, interceptors, replication groups, containers, nodes, *etc.*. Transitions rules further depend on the kind of faults that may occur and the redundancy provided by the middleware infrastructure that is used. At this point, the generation procedure must be customized accordingly.

   For active replication groups, for instance, and for the case of permanent faults the generic rule states that if a collaboration is in a state where n replicas are failed then the collaboration may get into a state where $n+m$ replicas are failed; the rate of such transitions equals to the arrival rate of the faults that caused the failure of the replicas. The m replicas may be instances of the same implementation of a middleware component, or instances of different dependent implementations of a middleware component.

Getting to our example, in the composition depicted in the lower diagram of Figure 2 (named composition A hereafter) a client request is first replicated. Each replicated request is encoded and sent to the target server replica. This composition is generic and can be used independently from whether different replicas belong to the same, or different security domains.

In the former case, however, it would be more efficient to use a middleware architecture like the one given in the upper diagram of Figure 2 (named composition B hereafter). In this composition a client request is replicated only after it enter into the common security domain. Even if composition B is more appropriate to be used in the case of replicated servers belonging to the same security domain it is expected to be less reliable than composition A as the connector between the client and the target security domain may be lossy. Hence, in cases where we have strong reliability requirements maybe it should be better to use A in the place of B.

By taking, however, a closer look into composition A we observe that security interceptors are used to encode replicated requests and we can distinguish two interesting cases at this point. First, the case where the replicated security interceptors are instances of the same implementation, based on functionality provided by single implementation of the CORBA security service. Second, the case where they are instances of different implementations, each one of which, is based on functionality provided by a multi-version implementation of the CORBA security service. The second case is less possible because middleware infrastructures usually do not come along with multi-version implementations of services.

In the first case, if a replica of the group fails due to a design fault, then it is most possible that all the replicas of the group shall fail at the same time due to the same design fault. In the second case, the previous is not likely to happen as different replicas are based on different versions of the security service. Based on the previous we generate three different state space models and use them as input to an existing reliability tool, named SURE-ASSIST [2], which is integrated into our environment..

Table 1 gives the results obtained from the tool. We can observe that the reliability for the case of composition A that is based on the multi-version security service is much better than the reliability of the

other two, which are actually close. Hence, it worths using composition A in place of composition B, mostly, if a multi-version implementation of the security service is provided.

In Table 1 we can further see statistics regarding the sizes of the generated models. For our simple example the middleware developer would have to build quite large models, which are now generated automatically by using the automated model generation procedures of the environment.

4 CONCLUSION

Existing middleware infrastructures provide highly flexible, reusable, functionality that can be composed in various ways towards satisfying certain functional and non-functional requirements. However, we believe that mastering and exploring this flexibility is not an easy task even for middleware experts. Based on the previous we are developing an environment that aims at facilitating the previous tasks. In particular, the environment provides automated support that allows middleware developers to automatically construct different possible middleware architectures satisfying certain functional requirements. Moreover, the environment provides automated tool support that facilitates the quality analysis of those different solutions against certain non-functional requirements.

References

[1] T. Margaria B. Steffen and M. von der Beec. Automatic Synthesis of Linear Process Models from Temporal Constraints: An Incremental Approach. In *Proceedings of the ACM-SIGPLAN International Workshop on Automated Analysis of Software (AAS'97)*, pages 127–141, 1997.

[2] S.C. Johnson. Reliability Analysis of Large Complex Systems Using ASSIST. In *Proceedings of the 8th AIAA/IEEE Digital Avionics Systems Conference*, pages 227–234, 1988.

[3] R. Kazman, S. J. Carriere, and S. G. Woods. Toward a Discipline of Scenario-Based Architectural Engineering. *Annals of Software Engineering*, 9:5–33, 2000.

[4] M. Klein, R. Kazman, L. Bass, S. J. Carriere, M. Barbacci, and H. Lipson. Attribute-Based Architectural Styles. In *Proceedings of the 1st IFIP Working Conference on Software Architecture (WICSA-1)*, pages 225–243, 1999.

[5] C. Kloukinas and V. Issarny. Automating the Composition of Middleware Configurations. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 241–244, 2000.

[6] C. Kloukinas and V. Issarny. SPIN-ing Software Architectures: A Method for Exploring Complex Systems. In *Proceedings of the 2nd IEEE/IFIP Working Conference on Software Architecture (WICSA-2)*, pages 67–76, 2001.

[7] J-C. Laprie. Dependable Computing and Fault Tolerance : Concepts and Terminology. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2–11, 1985.

[8] X. Qian M. Moriconi and R.A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.

[9] E. Marsden. Failure Modes Analysis of CORBA-Based Middleware - Preliminary Results. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'2001)*, 2001.

[10] B. Spitznagel and D. Garlan. A Compositional Approach for Constructing Connectors. In *Proceedings of the 2nd IEEE/IFIP Working Conference on Software Architecture (WICSA-2)*, pages 148–157, 2001.

[11] A. Zarras and V. Issarny. Automating the Performance and Reliability Analysis of Enterprise Information Systems. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, page to appear, 2001.

[12] A. Zarras, V. Issarny, C. Kloukinas, and V. K. Nguyen. Towards a base UML Profile for Architecture Description. In *Proceedings of the 1st ICSE Workshop on Describing Software Architecture with UML*, pages 22–26, 2001.