Valérie Issarny, Christos Kloukinas, and Apostolos Zarras

Systematic Aid for Developing Middleware Architectures

This development environment enables the specification, automated composition, and quality analysis of flexible, configurable middleware architectures, notably in distributed systems.

Middleware is necessary for developing distributed systems. Developers compose them from reusable services provided by standard or proprietary middleware infrastructures, including the Object Management Group's Common Object Request Broker (CORBA), Microsoft's Distributed Component Object Model, Sun Microsystems' Java Remote Method Invocation, and related services, to deal with nonfunctional requirements for distribution, security, transactional processing, and fault tolerance.[1] The development process has been made easier by following the object-oriented middleware paradigm toward the component-based middleware paradigm, which includes the CORBA Component Model, the Microsoft Transaction Server, and Enterprise JavaBeans.

Developers can ignore the sometimes considerably complex composition of middleware services. Instead, they build middleware components, deploying them in off-the-shelf middleware containers in order to customize the composition of middleware services. Containers range from base ones with standard and proprietary middleware infrastructures, to more complex ones from commercial vendors extending the semantics of the base containers. Base containers typically provide transparent persistent storage and transactional and secure access to components; complex containers further combine other proprietary services.

However, assembling off-the-shelf components into containers is less straightforward then it might appear. To be able to offer middleware containers, vendors have to design and implement architectures combining available middleware services into flexible and customizable structures. Middleware services are not monolithic; they consist of a number of elements that, used appropriately, provide certain nonfunctional properties. Combining more than one middleware service amounts to composing the elements of each service. In general, various methods are used to compose these elements to satisfy application nonfunctional requirements. The resulting compositions should be supported by the configurable middleware architecture provided by middleware container vendors, especially to programmers. Moreover, off-the-shelf middleware architectures should come with a quality assessment of the possible compositions they support. This assessment should give developers clues for selecting the most suitable compositions for their particular applications.

Addressing these issues, we have developed an environment that facilitates the design and quality analysis of flexible and configurable middleware

TERRY MIURA

[1] See www.omg.org/technology/documents/; www.microsoft.com/com/wpaper/compsvcs.asp; and java.sun.com/products.

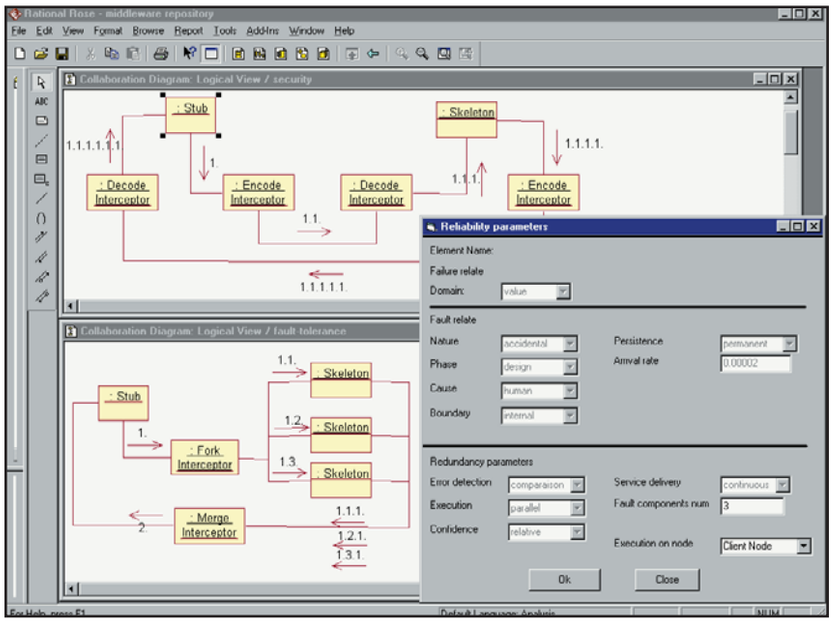

Figure 1. Configurations describing the interaction among middleware elements used for secure communication and fault tolerance.

architectures, providing three main features:

Architecture description language (ADL). Our ADL helps model middleware architectures.

Repository of architectural descriptions. Our repository is populated with architectural descriptions of middleware infrastructures that can include the specification of the basic services provided by the infrastructure (such as the CORBA ORB and Common Object Services), as well as related constraints. For each service, the architectural description also includes the specification of basic patterns describing the data and control interaction among the service's architectural elements toward provision of certain nonfunctional properties.

Automated tool support. Our automated support helps construct all possible valid compositions of a given set of interaction patterns, each describing how to use the elements of a middleware service in the interests of providing a particular nonfunctional property. It also helps generate performance and reliability analysis models for assessing and comparing valid compositions. These models serve as input to existing performance and reliability analysis tools integrated into the environment.

Modeling Middleware Architectures

Typical ADLs [8] provide basic modeling constructs for the specification of a number of architectural elements, including: components, or units of data or computation; connectors, or the interaction protocols among components; and configurations, or the assembly of components and connectors.

For modeling middleware architectures, our ADL also provides subtypes of the basic modeling constructs representing middleware-specific architectural abstractions like stubs, interceptors, containers, message-oriented connectors, stream connectors, and remote procedure call (RPC) connectors. Defining these abstractions is inspired by established middleware standards, including CORBA and the International Organization for Standardization's Reference Model for Open Distributed Processing.[2]

In order to render our ADL compatible with standard modeling notations, we have investigated the relationship of its basic constructs and standard elements of the Unified Modeling Language (UML).[3] As a result, we have defined components, connectors, and configurations as UML stereotypes extending, respectively, the semantics of UML subsystems, associations, and collaborations among types and instances of components and connectors. Defining constraints on the use of basic middleware elements involves the Object Constraint Language. Patterns of interaction are given in terms of configurations of instances of components and connectors.

Since the basic architectural constructs we use are extensions of standard UML elements, we use any existing UML modeling tool for specifying middleware architectures. Most of these tools are customizable, enabling integration of add-ins facilitating specification and validation of models, including user-defined stereotyped elements. For example, we designed and implemented an add-in that eases specification and validation of middleware architectural descriptions.

Figure 1 outlines how a developer can use a specific instance of our environment when using the Rational Rose UML modeling tool;[4] boxes represent component instances, and lines represent connector instances. The widget on the right side of the figure is part of the add-in we developed to enable the specification of certain component properties whose values

[2]See www.iso.ch:8000/RM-ODP.
[3]See www.omg.org/technology/documents/spec_catalog.htm#modelingspecs.
[4]See www.rational.com/products/softdev.jsp.

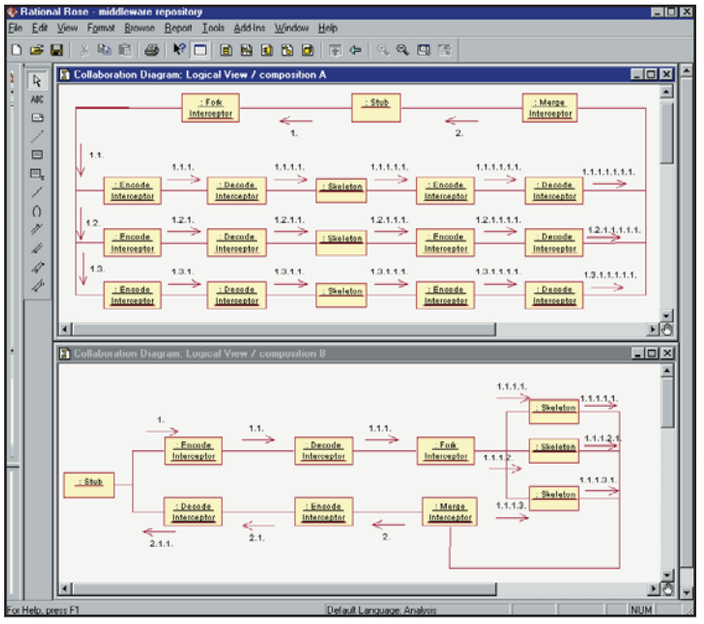

Figure 2. Two valid compositions of the configurations for fault tolerance and security.

are used in the quality analysis of the architecture.

According to the CORBA security service standard specification, client requests and server replies pass through several layers of interceptors from the sender to the receiver. One of them preserves integrity and confidentiality via existing cryptographic mechanisms. Figure 1 (upper configuration) abstractly describes the interaction pattern among these middleware architectural elements.

According to the standard for fault-tolerant CORBA, a CORBA-compliant infrastructure should support both passive and active replication styles. In the former, a client communicates with a replicated group of servers using simple Internet Inter-ORB Protocol invocations to the primary member of the group. In the latter, a client communicates with the group through a proprietary multicast group communication protocol. Figure 1 (lower configuration) abstractly describes the interaction pattern among elements realizing the group communication protocol.

Composing Middleware Services

Various approaches support the composition of software architectures. For example, [9] proposes that two architectures be composed by merging components from each of them; when the initial architectures do not share components, it further proposes a new "bridge" architecture containing a component from one architecture and another component from the other, enabling the two initial systems to communicate. Although this is a promising approach to constructing a single system, it is not effective for composing middleware architectures.

Applying the approach in [9] to the two patterns in Figure 1 (assuming the different interceptors share no features) gives developers building middleware architectures a pattern with a client stub and replicated skeletons and two different connection paths between them. One path enables secure communication; the other realizes the multicast communication protocol. Unlike composing architectures by merging components, developers need composition of previous patterns; the result is a single connection path providing a secure multicast communication protocol.

Another promising approach to composition, proposed in [7], involves a method for composing linear architectures, or architectures in which each component has a single input and a single output port. To compose a new architecture, the developer provides the linear architectures to be composed and a linear time temporal logic property that constrains the structure of the composed architecture. The method then constructs all possible compositions matching this property. However, not all middleware architectures are linear; for example, in Figure 1, the architecture for secure interaction is linear, but the one for fault tolerance is not.

Yet another approach, proposed in [10], introduces a set of operators that transform generic connectors (such as RPC) to incrementally add new nonfunctional properties. A basic problem with this approach for our goals in automating composition of middleware architectures is that the related transformations are built manually by the middleware architect and that their application results in a single composition. It would be more worthwhile to obtain all possible compositions automatically without having to describe all possible transformation operators. Such an automated convenience would allow middleware architects and designers to choose a suitable composition according to the nonfunctional requirements of the application they are building while exploring new and novel ways of using middleware services.

To enable the automated generation of composition, our environment includes a tool that automatically constructs all compositions of a set of interaction patterns, each describing the use of elements of a middleware service in order to provide a certain nonfunctional property. A composition is then selected by the middleware architect or designer according to the application's requirements, which are checked against the nonfunctional properties of the composed config-

urations via model-checking and quality analysis.

Automated composition of middleware architectures. To compose a set of interaction patterns, a tool has to find ways to connect the instances of the middleware elements so they offer all the nonfunctional properties of their respective patterns. In order to construct all possible compositions in the simple case of linear architectures (each component has one input and one output port), the tool examines all possible connections between the ports of the component instances used in the patterns. If the total number of component instances used in the set of patterns is *n*, then there are *n!* ways to connect them. In the more general case of nonlinear architectures, for every component instance having *m>1* output ports, the tool may need to create *m* component instances for each component instance used in the other patterns. For example, in the pattern for fault tolerance in Figure 1, because of the *ForkInterceptor* instance, which replicates request messages, the tool may need to create three instances for each middleware component instance used in the security pattern to secure each path leaving the *ForkInterceptor*; see the resulting composition A in Figure 2.

Cases	No. transitions	Reliability (upper bound)	Reliability (lower bound)
Composition A (multiversion security service)	48	0.80	0.79
Composition A (single-version security service)	24	0.74	0.72
Composition B	12	0.70	0.67

Results from the reliability analysis.

However, creating additional instances increases the complexity of the composition. Specifically, the upper bound, *M*, of the number of instances the tool may need to create is the product of the fan-out degrees, or number of output ports, of the middleware component instances in the initial patterns. Consequently, the number of cases the tool has to examine increases to *O((M * n)!)*. Since covering this state-space exhaustively is impossible in practice the tool uses the interaction information in the patterns to be composed to constrain the state-space. The tool constructs only the compositions that preserve the initial flows of interaction (such as those where messages sent by the *ForkInterceptor* are eventually received by the *MergeInterceptor*). This and other constraints derived from the initial patterns substantially reduce the number of different compositions the tool has to construct, making it feasible to construct all of them automatically [5].

When composing the patterns for security and fault tolerance, as in Figure 1, the tool constructs only the 23 different compositions abiding by the constraints, instead of trying to examine all *O((3*12)!)* possibilities, to connect the different component instances.

Once it constructs the structurally correct compositions, the tool checks which of them indeed provide the nonfunctional properties required by the middleware architect, model-checking them with the Simple Process Meta Language (PROMELA) Interpreter, or SPIN, model-checker. For this checking process, the tool uses PROMELA (the modeling language of SPIN) models of the individual architectural elements to construct and verify models for each of the compositions. The models of the individual elements are obtained by asking the architect to fill in automatically generated PROMELA skeletons when describing middleware architectures using our ADL. Generating the models from the skeletons is based on two generic mapping relationships:

- For each component, the tool generates a different type of PROMELA process, having as many communication channels as the component has input/output ports. This way, the architect provides the behavior of the process.
- For each different connector, the tool generates a different type of process, as before. These processes do not, however, have their own channels for communicating; instead, they use the channels corresponding to the component ports they connect.

See [6] for more on how the verification is performed, as well as the techniques needed to speed the verification process.

Analyzing Quality of Middleware Architectures

Pioneer work related to the quality analysis of systems at the architectural level includes Attribute-based Architectural Styles [4] and the Architecture Trade-off Analysis Method (ATAM) [3]. Except for specifying the architecture of the system, quality analysis at the architectural level involves the following steps:

- Identifying quality measures of interest, including reliability and response time;
- Specifying quality properties of the constituent architectural elements that might affect the quality measures, including failure rate, persistence of

faults, replication policy, and scheduling policy; and
- Specifying quality models based on the first two, such as Markov chains and queuing networks, which are solved or simulated to approximate the values of the quality measures.

Specifying quality models is complicated; according to [3], developing quality models accounts for about 25% of the total time needed to apply ATAM. The effort needed to specify quality models is even greater for typical middleware architects and designers with no experience in formal modeling and analysis methods.

MASTERING AND EXPLOITING THE FLEXIBLE AND REUSABLE FUNCTIONALITY INHERENT IN MIDDLEWARE INFRASTRUCTURES IS TIME CONSUMING AND PROBLEMATIC, EVEN FOR MIDDLEWARE EXPERTS.

To deal with the time-consuming complexity of specifying quality models, our environment includes a tool for generating quality models for performance and reliability analysis. The tool accepts (as input) patterns describing the interaction among elements provided by middleware infrastructures whose architectural description is stored in the repository. Middleware elements are characterized by properties whose values affect the values of the performance and reliability measures, as in the reliability properties associated with the components in Figure 1.

Automating quality analysis of middleware architectures. Building quality model generators involves defining the generic mapping relationships among the patterns describing the interaction among instances of middleware elements and models [11].

To evaluate reliability, we assume the basic measure is the probability that a pattern describing the interaction among middleware elements can take place during the lifetime of the application using the pattern. The pattern could fail if instances of middleware components, connectors, and nodes used in it fail due to faults causing errors in their state. (By definition, ADL components are associated with nodes on which the components are deployed.) Parsing the specification of the pattern allows the generation of a state-space model that can be used for approximating the values of the reliability measure. Such generation relies on the following generic mapping relationships:

Substates. A state in the state-space model is composed of substates, each representing the situation of an instance of a middleware component/connector/node used in the pattern.

Properties. The range of possible situations for a component/connector/node depends on the fault and replication properties characterizing the element; the generator must be customized accordingly.

States. A state in the state-space model is a death state if at least one of its substates represents a situation in which the corresponding component/connector/node has failed.

Transitions. Transition from a source to a target state represents a change in the situation of a component/connector/node.

Deriving transitions. Given a source state and the range of possible situations for a particular component/connector/node, all possible transitions can be derived automatically through the algorithm proposed in [2].

Performance-analysis models are generated in a similar way, but our environment supports the generation of queuing network models from patterns describing the interaction among middleware elements. Model generation relies on several generic relationships:

- Queuing network stations representing nodes and middleware connectors used in the pattern;
- Services provided by stations representing middleware components used in the pattern; and
- Data and control-flow information described in the pattern used to generate agents circulating around the stations asking for specific services.

In composition A in Figure 2, a client request is first replicated. Each replicated request is then encoded and sent to the replica target server. This composition is generic and can be used independently of whether different replicas belong to the same or to different security domains. If a replica belongs to the same domain, it would be more efficient to use a middleware architecture like composition B in Figure 2.

Here, a client request is replicated only after it enters into the common security domain. Even if composition B is more appropriate for replicated servers belonging to the same security domain, we expect it would be less reliable than composition A; less reliability can be assumed when message losses typically occur in the connector between the client stub and the target security domain. Hence, in cases involving strong reliability requirements, it may be better for the architect to use A instead of B, even if doing so is less efficient.

A closer look at composition A reveals that security interceptors are used to encode replicated requests producing two interesting cases regarding these interceptors:

- They are instances of the same implementation based on functionality provided by a single implementation of the CORBA security service.
- They are instances of different implementations, each based on functionality provided by a multiversion implementation of the CORBA security service.

In the former, if a replica of the group fails due to a design fault, the architect can conclude that all the replicas of the group are likely to fail due to the same design fault. In the latter, failure of all replicas is unlikely, as different replicas are based on different versions of the security service. Following the earlier discussion of middleware architecture quality analysis and the patterns in Figure 2, we generated three different state-space models, using them as input to the reliability tool SURE-ASSIST [1] we integrated into our environment.

The table lists the upper and lower bounds of the reliability of both composition A and composition B, as calculated by the tool. The results indicate it is worth using composition A in place of composition B if the architecture provides a multiversion implementation of the security service. Highlighting the gain from automatically generating reliability models, the table includes statistics regarding the number of state transitions in the generated models.

Conclusion

Existing middleware infrastructures provide highly flexible and reusable functionality that can be composed in various ways toward satisfying certain nonfunctional requirements. But mastering and exploiting this flexibility is time consuming and problematic, even for middleware experts. That is why we are developing an environment to facilitate the modeling, composition, and quality analysis of middleware components and architectures.

We've been experimenting with the environment on real-world cases in the context of the European Commission's Dependable Systems of Systems project. We are now interested in extending the approach toward the analysis of other aspects of large-scale distributed systems, including maintainability, openness, and scalability. C

References

1. Butler, R. The SURE approach to reliability analysis. *IEEE Transact. Reliab. 41,* 2 (June 1992), 210–218.
2. Johnson, S. Reliability analysis of large complex systems using ASSIST. In *Proceedings of the 8th AIAA/IEEE Digital Avionics Systems Conference* (San Jose, CA, Oct.). IEEE Press, Los Alamitos, CA, 1988, 227–234.
3. Kazman, R., Carriere, S., and Woods, S. Toward a discipline of scenario-based architectural engineering. *Annals Software Engin. 9* (2000), 5–33.
4. Klein, M., Kazman, R., Bass, L., Carriere, S., Barbacci, M., and Lipson, H. Attribute-based architectural styles. In *Proceedings of the 1st IFIP Working Conference on Software Architecture* (San Antonio, TX, Feb. 22–24). Kluwer Academic Publishers, Boston, 1999, 225–243.
5. Kloukinas, C. and Issarny, V. Automating the composition of middleware configurations. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering* (Grenoble, France, Sept. 11–15). IEEE Press, Los Alamitos, CA, 2000, 241–244.
6. Kloukinas, C. and Issarny, V. SPIN-ing software architectures: A method for exploring complex systems. In *Proceedings of the 2nd IEEE/IFIP Working Conference on Software Architecture* (Amsterdam, The Netherlands, Aug. 28–31). IEEE Press, Los Alamitos, CA, 2001, 67–76.
7. Margaria, T., Steffen, B., and von der Beec, M. 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* (Paris, Jan.). ACM Press, New York, 1997, 127–141.
8. Medvidovic, N. and Taylor, R. A classification and comparison framework for software architecture description languages. *IEEE Transact. Software Engin. 26,* 1 (Jan. 2000), 70–93.
9. Qian, X., Moriconi, M., and Riemenschneider, R. Correct architecture refinement. *IEEE Transact. Software Engin. 21,* 4 (1995), 356–372.
10. Spitznagel, B. and Garlan, D. A compositional approach for constructing connectors. In *Proceedings of the 2nd IEEE/IFIP Working Conference on Software Architecture* (Amsterdam, The Netherlands, Aug. 28–31). IEEE Press, Los Alamitos, CA, 2001, 148–157.
11. Zarras, A. and Issarny, V. Concurrency in dependable systems. Chapt. 7 in *Quality Analysis of Enterprise Information Systems,* A. Romanovsky and P. Ezhilchelvan, Eds. Kluwer Editions, 2002, 127–146.

Valérie Issarny (Valerie.Issarny@inria.fr) is senior research scientist on the Arles project in the Institut National de Recherche en Informatique et an Automatique, Chesnay Cedex, France.
Christos Kloukinas (Christos.Kloukinas@inria.fr) is a Ph.D. student in the Arles project in the Institut National de Recherche en Informatique et an Automatique, Chesnay Cedex, France.
Apostolos Zarras (apostolos.Zarras@inria.fr) is a post-doctoral researcher in the Arles project in the Institut National de Recherche en Informatique et an Automatique, Chesnay Cedex, France.