Automating the Composition of Middleware Configurations

Christos Kloukinas   Valérie Issarny

INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cédex, France
E-mail: Christos.Kloukinas@inria.fr , Valerie.Issarny@inria.fr

Abstract

A method is presented for the automatic construction of all possible valid compositions of different middleware software architectures. This allows reusing the latter in order to create systems providing a set of different non-functional properties. These compositions are constructed by using only the structural information of the architectures i.e.*, their configurations. Yet, they provide a valuable insight on the different properties of the class of systems that can be constructed when a particular set of non-functional properties is required.*

Keywords: *Middleware, Software configuration, Software architecture, Configuration composition.*

1. Introduction

Middleware infrastructures are recognized as providing powerful support for the construction of complex distributed software systems. However, the software development process is still error-prone since the application developer must master, possibly complex, mechanisms so as to select the needed middleware services and understand how to integrate them with the application components.

The Aster development environment[1] aims at easing the construction of distributed software systems out of such middleware platforms. It offers a number of tools for the systematic selection and integration of middleware components given the architectural description of a distributed application, including the application's non-functional requirements (*e.g.*, a tool for integrating the retrieved middleware configuration with the application through the generation of appropriate proxy objects) [6, 9].

A problem that rises when mechanizing the process of configuring a middleware is when the middleware must enforce different types of non-functional properties. An illustration of this is the combination of fault-tolerance and security properties, requiring composing middleware configurations respectively associated with these properties. The issue of composing software has formerly been addressed from a theoretical perspective, by examining the composition of software specifications (*e.g.*, see [2, 3, 8]). However, such an approach is known to be at the expense of automation. In this paper, we propose a more pragmatic approach, which consists of a tool that takes as input the configurations of the middleware architectures to be composed and computes all possible valid composite middleware configurations, thus exploiting the architectural styles of the initial systems. The developer can then select the composite configuration that best suits the application under construction, according to the application's non-functional requirements. Section 2 addresses further issues relating to the composition of middleware configurations, showing the benefits of handling it at the structural level. Section 3 then introduces our solution to the structural composition of middleware configurations, while we compare our approach with related work in Section 4 and offer some conclusions in Section 5.

[1] www-rocq.inria.fr/solidor/work/aster.html

2. Issues in composing middleware

The issue of composing software systems at the architectural level has been examined in [8] where two complementary kinds of composition were identified: *vertical* and *horizontal*. The former relates to the top-down refinement process. Horizontal composition is used to compose instances of architectures to form one large composite architecture. It is handled by the developer on a case-by-case basis according to a simple syntactic criterion *e.g.*, the composed architectures can share only components.

From the perspective of identifying a middleware configuration meeting application requirements, vertical composition serves refining those requirements into a concrete middleware configuration, which is the approach that is actually used in the Aster environment. However, vertical composition falls short when applications require different types of non-functional properties, as already suggested by the case-by-case approach to horizontal composition proposed in [8]. Another problem with this particular solution to horizon-

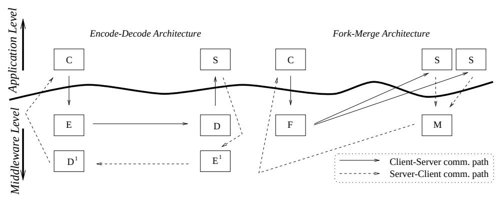
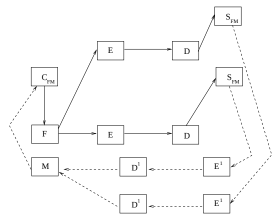


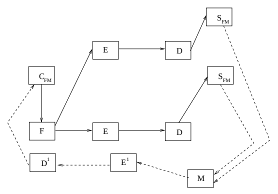
Figure 1. The Encode-Decode and Fork-Merge middleware configurations

tal composition is that, in general, it ignores various valid composition patterns. For instance, consider the two typical client-server-based middleware configurations depicted in Figure 1. The *Encode-Decode* (*E-D*) configuration on the left-side of the figure offers secure communication by encoding and decoding exchanged messages during a remote procedure call. The *Fork-Merge* (*F-M*) configuration on the right side of the figure offers fault-tolerance properties by replicating the server and sent messages, and then merging the results from the replicated servers through some voting algorithm. Various compositions may be considered (*e.g.*, see Figure 2). One that is trivial is depicted in Figure 2(a); it consists of encoding and decoding messages between the *Fork* and *Server* (*S*) components and between the *Server* and *Merge* components. However, as a less intuitive option, we may encode messages right after the server issues them and decode the messages right before their reception from the client (*e.g.*, see Figure 2(c)). Such an architecture requires processing encoded messages, which although not common, offers interesting capabilities [1].

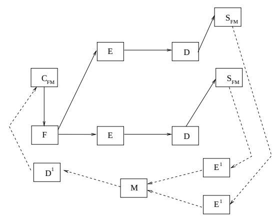
Having the necessary middleware configurations to enforce each of the targeted non-functional properties, the issue is then to compose those configurations. As raised above, there is a number of ways in which two middleware configurations may be composed. Our major design objective is to be able to automate the related process as much as possible. As a result, we have decided to not undertake a refinement-like approach, which relies on some theorem prover, since these are tools that need considerable guidance from users. Instead, we noted that this can be done in a first step at the structural level, without requiring the use of any formal tool. Basically, this first step is achieved by processing graphs encoding middleware configurations and its complexity depends on the number of nodes in the graphs. Once all the valid composite configurations are identified, the designer can easily select some of them, since configuration descriptions provide visual hints about the properties of the configuration. In a second step, the designer may assess the chosen configurations using some formal method, so as to identify the one that best suits the application's requirements.



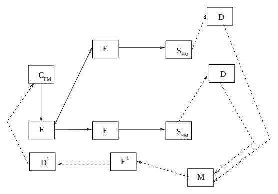
(a) A trivial, valid composition



(b) Another trivial, valid composition



(c) A valid, but less trivial composition



(d) An invalid composition

Figure 2. Some compositions of the Fork-Merge and Encode-Decode configurations

3. Structural composition of middleware

Typically, a *middleware* configuration consists of the following elements:

- The *middleware components*, that represent the middleware services used for enforcing the non-functional properties associated with the given configuration;
- The *generic components* that represent the application components where the middleware configuration plugs-in;
- The *connections* among the components, which represent the base messaging protocol used for interaction among them.

Consider now the composition of two middleware configurations (*e.g.*, the ones depicted in Figure 1). One trivial solution is to consider that one of the two configurations is to be used for carrying out any of the interactions of the other. However, this simple scheme is not well suited because even though any middleware configuration obeys the architectural style of the associated middleware infrastructure from the perspective of the application's components, the interactions that are internal to the middleware configurations do not necessarily follow the same style, due, in particular, to performance concerns. For instance, both configurations of Figure 1 obey the client-server architectural style from the viewpoint of the application components, as traditionally offered by most existing middleware infrastructures, while the middleware components interact according to a simple message-passing architectural style.

Therefore, when composing we must respect both the architectural style used internally by the middleware configuration (*e.g.,* message-passing in our examples), as well as the one exported to the application layer (*e.g.,* client-server in our examples). Indeed we create systems, where the second architecture is applied *i.e.,* its generic components are mapped to, to some components of the first. In doing so, the only assumptions we make are that middleware components can in general process messages arriving from other middleware components (since these were designed to be reusable in the first place) and that application components can receive messages only if these are "generic friendly" *i.e.,* they were received by some generic component in one of the initial architectures.

The idea behind the proposed algorithm becomes more clear if we assume that the two architectures A and B to be composed are network stacks and think of them as strings, $\alpha = a_1 \cdots a_n$ and $\beta = b_1 \cdots b_m$ respectively, where each letter stands for a component. Then, our goal is to create all possible strings $\alpha \otimes \beta = \gamma = c_1 \cdots c_{n+m}$, such that $c_j \in \alpha \vee \beta$ and the order between different letters of the same initial string *i.e.,* the order in the communication path, is preserved. If we think of this as looking for n places in γ to place the letters of α, this lets us calculate the number of different possibilities as $\binom{n+m}{n}$. The results can be constructed by creating a binary tree, where at each node we choose the next letter either from α or β.

With configurations, however, there are additional constraints we must abide by, in order to produce valid results for $A \otimes B$. First, the generic components of B must be mapped to some components of A, so as to identify the components of A for which the properties of B will be provided (thus the operator $\otimes$ is *non-commutative*). Therefore, when the next component of B during the tree traversal is generic, we can choose it only if the last component chosen (the parent) was a component of A. This way, we map B's generic component to that component of A. In addition, when we reach a leaf of the tree (which is a node where either one of the architectures has no more components left to choose from) we check to see whether there are still generic components in B that have not been chosen/mapped. If so, we consider this leaf/solution as an invalid one. A second constraint is that we cannot allow generic components of A to be found between middleware components of B, since generic components should not receive messages initially meant to be received and processed by some middleware component. Therefore, when the parent of some node is a middleware component of B while the next component of A is a generic one, we allow the latter to be chosen, only if, originally, the former was sending messages to a generic component of B, since we assume that these messages were "generic friendly". A last constraint is that order in the communication path should be preserved. That is, if components a and b were communicating directly in one of the initial architectures ($a \rightarrow b$), then there should be a path from a to b in each of the results, so that messages/data have eventually the same treatment in both architectures. This, however, is satisfied by the manner itself in which we construct the results. A final point that needs clarification is the choice of the first/root components. These are given for each of the initial architectures as the components from which the enforcement of the respective non-functional property associated with the particular configuration is initiated. Typically, this is the generic *Client* component in a client-server-based configuration. For each of the results we assign as root component that of the first configuration *i.e.,* of A [2].

Figure 2(d) shows a configuration that is not valid because it contains a path from *E* to *D* which includes the generic node S_{FM}, even though *E* and *D* communicated directly originally. Thus, this configuration does not conform to the second constraint.

The above constraints lead to a significant decrease in the number of possible solutions. For example, for the E-D and F-M architectures (6 and 4 components respectively) it produces only 26 valid composite configurations, while $\binom{6+4}{6} = 210$. In addition to these constraints, the developer may reduce even more the number of solutions by abstracting away components. For example, if the *Merge* component should always be found at the client's side, then one can abstract those two into a new client component that receives the replies from the servers and merges them itself.

Up to this point, we have only discussed the composition of two configurations. The composition of any number of configurations is achieved by iteratively applying the composition algorithm. Also notice that our algorithm does not cope with nodes having multiple *heterogeneous* outgoing arcs. At this time, we consider only multiple *homogeneous* arcs that lead to the same type of nodes (*e.g.,* see the links from the *F* component of the *F-M* configuration), which are actually handled as a single link *i.e.,* the nodes that are pointed to may be abstracted as a single complex node (*e.g.,* this case is illustrated in Figures 2(a), 2(b) and 2(c) with the two links from *F* to S_{FM} that are both enriched with the *E* and *D* components). We are currently extending our algorithm so as to cope with nodes having heterogeneous outgoing arcs, as well as, investigating ways to allow users describe additional constraints to be met *e.g.,* that two particular components should (or should not) be directly connected.

4. Related work

To the best of our knowledge, the computation of composite configurations has not been examined in the past.

[2]An implementation of this algorithm can be found at: `www-rocq.inria.fr/solidor/code/comp-algo-single.lisp`

Related work is mainly concerned with the composition of software systems at the specification level, hence leaving few opportunities for automating the process. The work that is the most related to ours *i.e.*, [8], has already been addressed in Section 2; this section identifies other work done on the composition of specifications of software systems. We in particular identify work appertained to software development processes, which prescribes defining multiple architectural views for the software system in order to address separately the concerns of the various stake-holders (*e.g.*, end-users, engineers, developers). In this direction, we find the work of [7] that introduces the "4+1" views of a software system architecture. The four system views (logical, process, development and physical) are linked together through use-case scenarios (*i.e.*, the "+1" view). Multiple-view descriptions of a software system were also considered in [4, 5]. This area of research work relates to ours in that it is concerned with the decomposition of the system's software specification in terms of various architectural views. However, the architectural views we consider are at a higher level of design, and all relate to the middleware architectural styles underlying the execution of distributed applications. Specifically, the software system "views" we focus on prescribe the middleware configurations to be used for enforcing given types of non-functional properties, which is to be composed with similar "views". In general, our work is complementary to the above references in that it offers methods and tools for helping the designer in reusing existing middleware platforms for the design of a software system complying with the architectural views that are set up during the design phase and that integrates non-functional considerations (*e.g.*, the process and physical views in the "4+1" views approach).

Composing components offering various properties (or *features*) has also been investigated in [10] for the specific case of telecommunication applications, where components can be assembled according to the *pipe-and-filter* architectural style. Even though features are usually self-contained and independent from the others, inconsistencies do arise and these are left to be resolved by the developer. Our context, however, is broader and cannot be reduced to simple horizontal compositions chaining middleware configurations. In fact, pipe and filter compositions rarely appear to be valid middleware compositions.

5. Conclusions

We have shown how it is possible to reuse middleware architectures providing specific non-functional properties to create systems that provide a multitude of these. The proposed method creates all possible systems that provide a particular set of non-functional properties by using just the structural information that is part of the initial middleware architectures *i.e.*, their configurations. This allows it to be automated and does not require, at least at this level, use of tools such as theorem provers that need substantial user guidance.

It is our belief that it can also help at the discovery of incompatibilities between middleware components, since the designers of the latter can easily visualize all possible ways their components may be used with particular classes of other middleware and either remove these problems early on, or warn their users of specific cases for which their product is not well suited for. This can lead to the construction of components that are easier to understand and are more trusted by their users, since all the possible ways in which they could be used are well identified as are the cases for which their use is inappropriate.

We are currently investigating integration of our solution to the composition of middleware configurations within the Aster environment so as to ease the assessment of the computed composite configurations regarding provided non-functional properties, as well as new versions of the algorithm that would allow it to process more complex configurations.

Acknowledgments. This work is partially funded by the IST DSoS project.

References

[1] M. Abadi, J. Feigenbaum, and J. Kilian. On Hiding Information from an Oracle. *Journal of Computer and System Sciences*, 39(1):21–50, 1989.

[2] M. Abadi and L. Lamport. Composing Specifications. *ACM TOPLAS*, 15(1):73–132, 1993.

[3] M. Abadi and L. Lamport. Conjoining Specifications. *ACM TOPLAS*, 17(3):507–534, 1995.

[4] A. C. W. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multiperspective Specifications. *IEEE TSE*, 20(8):569–578, 1994.

[5] P. Fradet, D. L. Metayer, and M. Perin. Consistency Checking for Multiple View Software Architectures. In *ESEC/FSE'99*, pages 410–428. Springer, 1999.

[6] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-based Development Environment: Experience with the Aster Prototype. In 4^{th} *Int. Conf. on Configurable Dis. Sys.*, pages 275–283, 1998.

[7] P. B. Kruchten. The 4+1 View Model of Software Architecture. *IEEE Software*, 12(6):42–50, 1995.

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

[9] A. Zarras and V. Issarny. A Framework for Systematic Synthesis of Transactional Middleware. In *Middleware98*, pages 257–274, 1998.

[10] P. Zave and M. Jackson. A Component-based Approach to Telecommunication software. *IEEE Software*, 15(5):70–78, 1998.