Towards Systematic Synthesis of Reflective Middleware

Petr Tůma, Valerie Issarny, Apostolos Zarras

INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{tuma, issarny, zarras}@irisa.fr
http://www.irisa.fr/solidor/work/aster.html

Abstract. In this paper, we present a method for systematic synthesis of middleware based on the meta-level requirements of the application that stands on top of it. Particular attention is paid to the ability to accommodate evolving requirements of a running application.

1 Introduction

Following the definition of programming languages supporting reflection [3], we say that a middleware supports *reflection* if the application that lies on top of it is provided with means for reasoning about the middleware properties and for customizing the properties as necessary. Many middleware infrastructures, such as FlexiNet and reflective Orbs, expose functionality in a way that can be used to customize the properties. This process is called *reification* [2] of the middleware functionality. Reification alone, however, does not equal reflection. What is missing are the means for reasoning about the properties of the middleware. The reasoning itself is left to the application designer, who is free to make unconstrained changes to the reified functionality and has to deduce the impact of the changes on the middleware properties himself. We remedy this drawback by designing a method for the systematic synthesis of middleware based on the application requirements, detailed in [4] and outlined in Section 2. Here, we concentrate more on how to use the systematic synthesis to accommodate changes in the requirements of a running application, as detailed in Section 3.

2 Systematic Middleware Synthesis

The systematic synthesis of middleware uses an architectural description of the application consisting of two views. The *structural view* describes interconnection of the individual components of the application; the *property view* describes meta-level properties of the middleware that implements the interconnections. The meta-level properties are described formally using temporal logic, a theorem prover is used to check the relationship between properties.

The available middleware services are stored in a middleware repository together with descriptions of the properties they provide. Based on the specification of requirements on the middleware properties, the appropriate middleware services are retrieved from the repository, assembled into a middleware and incorporated into the application [4].

3 Dynamic Reflection

Here, we extend the systematic synthesis of middleware with the ability to build middleware that can be dynamically exchanged so as to reflect changes in the requirements of a running application. The problem of exchanging the middleware is twofold. Immediately visible are the technical issues related to dynamic loading and unloading of the middleware code, where many solutions have been proposed. Less visible is the issue of what impact the dynamic change of the middleware code has on the properties provided by the middleware.

At the time a new middleware M' is to replace the old middleware M, there might be requests issued through M for which the application requirements are not satisfied yet. Examples of these requests include an unfinished remote procedure call when reliable delivery is required, or an open distributed transaction when some of the ACID transaction properties are required. The new middleware M' must guarantee that the requirements $R(req)$ valid at the time these *pending requests* were issued will be satisfied. During system execution, the information necessary to complete pending requests is a part of the middleware state. The new middleware should start from an initial state that contains this information and should be able to satisfy requirements related to the pending requests. Given a specific mapping $Map(\sigma)$ between the states of the old and the new middleware, we define a *safe state* σ_M in which it is possible to perform the exchange while satisfying the requirements:

$$\begin{aligned} &SafeState(\sigma_M, M, M', Map) \equiv \\ &\quad [\sigma_M] \wedge \forall req \mid Pending_M(req) : [Map(\sigma_M)] \Rightarrow R(req) \end{aligned}$$

A safe state is defined in relation to the ability to map the state from the old to the new middleware. When no state mapping is available, we refine the $SafeState$ predicate into a stronger criterion, $IdleState$, defined only with respect to the old middleware properties. In an *idle state* [1], no requests are pending and hence no state needs to be mapped:

$$IdleState(\sigma_M, M) \equiv [\sigma_M] \wedge (\forall req : \neg Pending_M(req))$$

Based on a straightforward utilization of the safe state definition, the general strategy of the exchange is to reach the safe state of the middleware, block incoming requests to remain in the safe state during update, exchange the middleware implementation, and unblock incoming requests. To detect whether the middleware reached a safe state, an idle state detection code is incorporated into it at the time it is being synthesized. The definition of the idle state makes this code independent of the middleware implementation; it is retrieved from the repository based on the middleware properties only. During exchange, this code is used to determine when it is safe to perform it; alternatively, a safe state detection code specific to the particular update can be installed.

It is generally not guaranteed that the middleware will reach a safe state within finite time during normal execution. We therefore selectively block requests from the application, so as to prevent activities that do not participate in driving the middleware into a safe state from issuing requests that would keep the middleware away from the safe state. It can be shown that the decision whether to block a request depends only on the safe state used during the update. This decision is therefore taken by the safe state detectors associated with the update.

For the purpose of the exchange, the middleware is separated into a static and a dynamic part. The static part of the middleware contains the proxy through which the application accesses the middleware, and the code for blocking requests. The dynamic part contains the safe state detection code and the middleware itself, both retrieved from the repository based on the required properties. The exchange is directed by a coordinator component, responsible for exchange within the scope of the changed property. When requested to perform an exchange and given the new code of the dynamic middleware parts, the coordinator instructs the static middleware parts to block the requests as described above. After a safe state is reached, the coordinator directs the middleware to unload the existing dynamic parts and install the new code.

4 Conclusion

The approach presented in this paper tackles the problem of changing middleware properties in a running application. Its advantage is in synthesizing the middleware systematically based on the required properties, as opposed to only exposing the functionality through reification. The basic concepts of the approach were prototyped using several CORBA platforms and the STeP theorem prover. The current work focuses on improvements related to granularity and timing of the middleware exchange, and on using the approach to synthesize adaptive middleware.

Acknowledgement: this work has been partially supported the C3DS LTR Esprit project.

References

1. J. Kramer and J. Magee. The Evolving Philosophers Problem. *IEEE Transactions on Software Engineering*, 15(1):1293–1306, November 1990.
2. J. Malenfant, M. Jacques, and F.N. Demers. A Tutorial on Behavioral Reflection and its Implementation. In *Proceedings of REFLECTION '96*. ECOOP, 1996.
3. B. C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, MIT, 1982. Available as MIT Techical Report 272.
4. A. Zarras and V. Issarny. A Framework for Systematic Synthesis of Transactional Middleware. In *Proceedings of* MIDDLEWARE *'98*, pages 257–272. IFIP, Sept 1998.