Formalising Dynamic Composition and Evolution in Java Systems

Claus Pahl

Dublin City University, School of Computer Applications
Dublin 9, Ireland
cpahl@compapp.dcu.ie

Abstract. The Java platform allows the dynamic establishment and closure of connections and also the composition and customisation of components at deployment time. In order to guarantee reliability and maintainability in dynamic evolving systems, we take a process-oriented view on composition and interaction. This is supported by a contract concept to formalise matching of suitable service provider and requestor.

1 Motivation

Forming a contract between service provider and service requestor can constrain the invocation of remote or unknown methods [1], leading to more reliable systems. Since dynamic loading of classes is possible in Java, objects can interact via RMI, and beans can use each others services [2] - possibly assembled and customised at deployment time -, respective means to control the use of services should be in place. A contract-based composition framework is sufficient for static systems, but systems do evolve over time. Requirements change and force contracts to be renegotiated. This can be addressed by embedding a concept of contracts into a model of change. Reasoning about the impact of dynamic composition and change is important to achieve reliability for evolving systems. We adapt a process calculus to capture the establishment and release of contracts.

2 Contracts and Connectors

The π-calculus shall be used to model the process of establishing contracts and connections between components. The π-calculus offers means to specify communication between agents in a distributed environment. Objects communicating through RMI and beans are agents in this sense. Modelling the process of change using a process calculus is justified by a similarity between mobility and evolution. Mobility in the π-calculus is defined as a change of neighbourhood, i.e., a change of the links that an agent has with its environment. In the same way evolution might require changes in connections between interacting objects or between beans. Requirements of a client, formulated using pre- and postconditions, need to be satisfied, or matched, by a service provider, formalised using the refinement calculus. Interfaces can be used to form contracts between a server

component and a client component. A requested and a provided method have to be matched based on their specifications (pre- and postconditions) to form a contract. The matching construct is refinement $\sqsubseteq$. The provider needs to satisfy the needs of the requestor, i.e., a provided method n should refine the requirements of m:

$$m \sqsubseteq n \stackrel{\triangle}{=} pre(m) \rightarrow pre(n) \wedge post(n) \rightarrow post(m) \tag{1}$$

The process of matching and creating a connector is described by:

$$\textsc{Req}\ \overline{cC}\langle m\rangle.C' | \textsc{Prov}\ cC(n).P' \stackrel{m \sqsubseteq n}{\longrightarrow} \textsc{priv}\ m\ (C' | P'\{^m/_n\}) \tag{2}$$

This rule is constrained by the refinement $m \sqsubseteq n$, i.e. it matches a service request m and a provided service n. This request is handled on a contract channel cC. This step establishes a (private) connection m between the provider and its client. Interactions between them, e.g. invocations of remote methods, can now be executed via the connector m. This rule can model contracts between client and server using RMI or between beans assembled to larger components. Connectors occur in two forms in Java. Firstly, as a remote computation, i.e., a service channel is used to invoke a remote method. Secondly, as a local computation, i.e., a data channel is used to load the class which contains the code to be executed. Connectors are an abstraction to capture *remote* and *mobile* code.

In order to formalise the constraint language within the dynamic framework, we need to see objects as entities with internal structure. We use hidden algebras to define semantical structures, embedding this into dynamic logic.

3 Management of Change

This framework for change and evolution in Java can be expanded into concepts to determine the effects of change and to manage evolving systems. Both specifications of service requests and available services might change due to changes in the overall requirements or the environment. Changes in one component might force changes in other components - change is propagated. A framework based on matching and internal correctness conditions can help to determine the effects of change. This framework can be defined based on the dynamic logic semantics used to embed pre- and postconditions. Matching is used to determine the effect of change to contracts. Internal component correctness relations form a measure for the effect of contract changes on a component implementation.

References

[1] L.F. Andrade and J.L. Fiadero. Interconnecting Objects via Contracts. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[2] S. Cimato and P. Ciancarini. A formal approach to the specification of java components. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Tech. Rep. 251, University of Hagen, 1999.