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. A variety of Java constructs involve an idea of time: dynamic establishment and closure of connections or the composition and customisation of components. In order to guarantee reliability and maintainability in dynamic evolving systems, we will take a process-oriented view on composition and interaction. This will be supported by a contract concept to formalise matching of suitable service provider and requestor.

1 Introduction

A common problem in complex dependable systems is to ensure that a method that is called actually provides the expected functionality. Forming a contract between service provider and service requestor can constrain the invocation of remote or unknown methods. In particular since dynamic loading of classes is possible in Java, respective means to control the use of services should be in place. This also applies to the assembly of beans – Java's component approach – since interaction is the composition principle. Contracts describe an agreement between service provider and client that can be checked statically or dynamically.

These compositions - objects interacting via RMI or beans using each others services - are compositions in *space*. Various approaches exist to describe this concept, e.g. [1, 2, 3]. However, several Java composition constructs also involve a notion of *time*. Classes can be loaded dynamically. Beans can be assembled and customised at deployment time. RMI incorporates the concept of a lease - a contract between objects that can expire and that can be renewed. 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 problem shall be addressed by embedding a concept of contracts into a model of change. We shall formulate a process-oriented model of composition based on the π-calculus [4] serving as a coherent semantical foundation for the variety of Java constructs involving a notion of time. Determining and reasoning about the impact of dynamic composition and change is of major importance to achieve reliability and maintainability for evolving systems. We use widely accepted formalisms for the specification of services, matching of services, and contracts. We use the pre- and postcondition technique, embedded into a dynamic logic [5, 6, 7], to specify services and contracts [8, 9, 10]. This forms the foundation for a matching construct based on refinement [11, 12].

2 Dynamic Composition and Evolution

Two main features of Java are portability and mobility of code. Objects can be passed around and loaded dynamically. In particular in combination with dynamic composition and customisation of components, this can affect the reliability of a system. *Java beans* are components that can interact with another. Beans can be customised at deployment time. RMI is the underlying distribution mechanism for beans. Enterprise Java Beans are beans for a server-side environments, which are created, configured and executed within containers that handle all interaction between the bean and its environment. The communication establishment using RMI could be constrained by contracts. A new feature for the RMI API is a *lease*. A lease is a mutual agreement between two objects for a period of time. These objects can negotiate and establish contracts for the use of resources. These contracts - leases - can expire, be cancelled, and can be renewed. The formulation of a process model for contracts and composition is therefore an adequate approach.

The π-calculus [4] 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. Both objects communicating through RMI and beans using services of other 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. Here, the interconnection between agents shall be constrained. Requirements of a client, expressed using pre- and postconditions, need to be satisfied by a service provider. Matching is the determination of the satisfaction of a required service. The π-calculus provides a theory of process equivalence - bisimilarity - based on observable process behaviour. Pre- and postconditions are additional properties and their matching needs to be supported by another theory: the refinement calculus. Our objective is to adapt the π-calculus [13, 14] in order to capture the establishment and release of contracts and connectors in evolving systems. We aim at a coherent formal basis for the variety of Java constructs with a notion of time.

3 Contracts

Interfaces describe entry points to a component. Specifications of these entry points can be used to form contracts between a server component and a client component. We assume that a client component (any object or bean) needs services of a server component. Two channel types are needed for data (code, events, etc.) and services, *data* and *serv*, respectively. A service channel realises the invocation of a remote method. A sorting discipline will ensure proper use of the channels. We need to choose a suitable provider candidate:

$$\textsc{Choose}\ \overline{sC}\langle cC\rangle.cC(x).C|\textsc{Offer}\ sC(y).\overline{y}\langle\epsilon\rangle.P \longrightarrow C|P \tag{1}$$

An initial output $\overline{sC}\langle cC\rangle$ of a channel name cC on channel sC by the client is received $sC(y)$ and answered $\overline{y}\langle\epsilon\rangle$ with an empty token by a suitable service provider P, creating a private contract channel cC for further use. Input and output actions can be successfully matched and, therefore, the agents - composed in parallel - transfer to the next state $C|P$. We have annotated the basic actions in order to illustrate their context.

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

$$m \sqsubseteq n \overset{\triangle}{=} pre(m) \rightarrow pre(n) \wedge post(n) \rightarrow post(m) \tag{2}$$

Matching - the next step - is formalised by the Match-rule

$$\textsc{Req}\ \overline{cC}\langle m\rangle.C' | \textsc{Prov}\ cC(n).P' \underset{m:serv}{\overset{m\sqsubseteq n}{\longrightarrow}} (C'|P') \tag{3}$$

and constrained by the sorting constraint $m : serv$ and the refinement $m \sqsubseteq n$. The following step establishes a (private) connection m - the Establish-rule:

$$\textsc{Estb}\ \overline{cC}\langle m, D\rangle.C'' | \textsc{Estb}\ cC(n,d).P'' \underset{m:serv}{\overset{m\sqsubseteq n}{\longrightarrow}} \textsc{priv}\ m{:}iC\ (C''|P''\{^m/_n\}) \tag{4}$$

where D is a deployment descriptor object. These rules can model contracts between client and server using RMI or between beans assembled to larger components. Other rules include for instance closing or re-negotiating a contract.

In order to formalise the constraint language within the dynamic framework, we need to see objects as entities with internal structure, e.g., in the style of labelled transition systems. We follow the hidden algebra approach [15] to define semantical structures for the refinement calculus, which is embedded into dynamic logic – and not predicate transformers [11]. We do not describe this in detail here, since our focus shall be on the dynamic calculus, e.g. [3].

4 Connectors

Connectors form an interconnection between two objects. They 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*. Beans, for instance, communicate with their environment using event handling. Other beans can register with a bean and will be notified in case an event occurs.

$$\textsc{EvReg}\ \overline{m}\langle \mathit{self}\rangle.C' | \textsc{EvReg}\ m(obj).P' \longrightarrow \textsc{priv}\ r{:}rC.\ (C'|P') \tag{5}$$

This establishes a reply-channel r of type rC for event notifications.

$$\textsc{EvNot}\ \overline{r}\langle eObj\rangle.C' | \textsc{EvNot}\ r(eObj).P' \longrightarrow C'|P' \tag{6}$$

As a result of an event, a notified client might request server methods. The interaction between the client C and server P

$$\textsc{Write}\ \overline{m}\langle a\rangle.C'''|\textsc{Read}\ m(x).P''' \xrightarrow[m:serv]{} C'''|P''' \tag{7}$$

can happen if permitted by the sorting rules. Further rules are necessary to describe the concurrent interaction of one component with several others. A rule allowing to reply to a service request can also be introduced.

5 Determination and Management of Change

Based on our formal framework for change and evolution in Java, we outline how this framework can be expanded into concepts to determine 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 problem impacting change is that resources can be shared. This includes sharing server functionality. Changes in shared server implementations (e.g. EJBs) have, therefore, a high impact on other components.

A framework based on matching and internal correctness conditions can help to determine the effects of change [3]. This framework is 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. Relations can be defined between import and body, or between body and export of a component. The relations can be defined using constructs such as model classes and relations between them.

An improvement might be achieved by adding glue-code between a changed server and a client in order to repair some of the change effects.

6 Conclusions

We have presented a formal model which captures concepts of dynamics and change in Java systems. It gives formal semantics to Java constructs with a notion of time, such as dynamic loading of classes, RMI interconnections and leases, and bean assembly, customisation and interaction. The key characteristics is a process-oriented view, into which a contract-concept for service matching is integrated. Our framework can be used to reason about system properties, e.g., to reason about change and effects on consistency, i.e., the determination of effects and preservation of consistency, in order to increase reliability and maintainability. It can provide the foundations for a formal development method.

The pre- and postcondition technique is only one possible (but very popular) form of expressing contracts. Other forms, for example including more than pure functional abstraction, can be considered in the future. Another important future aspect is the inclusion of our formal framework into a development tool. Examples for these environments are ESC Java [16] and the KeY-tools [17].

References

[1] 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.

[2] G.T. Leavens and M. Sitamaran. *Foundations of Component-Based Systems.* Cambridge University Press, 2000.

[3] C. Pahl. Modal Logics for Reasoning about Object-based Component Composition. In *Proc. 4rd Irish Workshop on Formal Methods, July 2000, Maynooth, Ireland.* BCS, eWiC series, 2000. (to appear).

[4] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[5] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier Science Publishers, 1990.

[6] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential logic. In S.D. Swierstra, editor, *Proc. ESOP'99 European Symposium on Programming Languages and Systems.* Springer-Verlag, 1999.

[7] B. Beckert. A dynamic logic for java card. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs.* Technical Report 269, Fernuniversität Hagen, 2000.

[8] J.B. Warmer and A.G. Kleppe. *The Object Constraint Language : Precise Modeling With UML.* Addison-Wesley, 1998.

[9] 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.

[10] G.T. Leavens and A.L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language.* Springer Verlag, LNCS 1723, 1999.

[11] R.J.R. Back and J. von Wright. *The Refinement Calculus: A Systemtic Introduction.* Springer-Verlag, 1998.

[12] M. Büchi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP '97.* Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[13] C. Pahl. Components, Contracts and Connectors for the Unified Modelling Language. In *Proc. Symposium Formal Methods Europe 2001, Berlin, Germany.* Springer-Verlag, LNCS-Series, 2001.

[14] M. Lumpe, F. Achermann, and O. Nierstrasz. A Formal Language for Composition. In G.T. Leavens and M. Sitamaran, editors, *Foundations of Component-Based Systems.* Cambridge University Press, 2000.

[15] J. Goguen and G. Malcolm. A Hidden Agenda. *Theoretical Computer Science*, 2000. Special Issue on Algebraic Engineering .

[16] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. Research report 159, Compaq Systems Research Center, 1998.

[17] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The key approach: Integrating object-oriented design and formal verification. Technical report 2000/4, University of Karlsruhe, Department of Computer Science, 2000.

Background and Expectations

Background

My research background lies essentially in software engineering, in particular foundations and formal aspects of software engineering. Main areas of interest during the last years have been state-based specification and implementation (in an algebraic style, using modal logics as the specification framework), and module and component languages. Recently, I have been looking at formal approaches for components and component composition. My interest has always been lying in integrating and bridging specification/design and implementation.

I have been involved in several national (Germany, Ireland) and international (EU) projects focusing on language semantics, design and implementation. Research topics have been, among others, set-theoretic programming and modular language semantics. The former involved the implementation of concepts of set theory in a programming language. The latter has focused on providing formal semantics or semantical concepts for the modular definition of programming and specification language, e.g., allowing languages (and their specifications) to be integrated or interfaced.

Java is a language that I am using in some of my courses. The experience of using Java in larger student projects shows the need of a clear understanding of fundamental concepts. I have been using Java in team-oriented projects that were supposed to implement e-Commerce systems. The development of software in distributed and heterogenous environments (such as e-Commerce systems) by teams of programmers shows the need for a component concept for Java. This is not only for educational purposes, but this teaching experience clearly motivates and supports my current research focus.

This paper can be seen as a contribution to an effort of building up a research group with a focus on component-based software engineering. My aim is to bring researchers from local universities (the greater Dublin area) together. Other researchers here have been working on subject-oriented design and implementation, or have an interest in concepts and technologies for components in heterogeneous environments. Initial funding is available. In the future this group shall address composition concepts and in particular the effect of change on component composition. Applications of results to Java are envisaged.

Expectations

My expectations in relation to the workshop would lie in the discussion of research ideas related to component-based software development in the Java context, and, if possible, to establish contacts to other researchers.

In general, the definition of strategic research directions in this and related areas would be of importance. In particular the problem of integrating and interfacing specification and implememtation should be addressed at the workshop. A straightforward combination could be to focus in OCL as the integrative means for UML and Java.

A second personal aspect is the issue of integrating the ideas presented in the paper into a suitable development environment. Discussions at the workshop could clarify this issue.