COMPASS: Tool-supported Adaptation of Interactions

Dirk Heuzeroth

Universität Karlsruhe
Program Structures Group
76133 Karlsruhe, Germany
E-mail: heuzer@ipd.info.uni-karlsruhe.de

Abstract

This paper presents an aspect-oriented approach and tool to consistently exchange and adapt interactions among software units. This is done by first identifying components, their interactions and interaction patterns. Second, the identified interaction points of components are represented as aspect-oriented ports *encapsulating the source elements related to the interaction. The interactions themselves are represented as first-class entities in the form of* aspect-oriented connectors *connecting the ports of components. These component, port and connector entities constitute an architectural model. Third, the developer reconfigures and adapts interactions exchanging the port and connector entities. This triggers corresponding source code transformations realized as meta programs using the Recoder tool. This approach is implemented as the COMPASS (COMPosition with AspectS) tool, that can currently analyze and transform Java source code based on the infrastructure provided by the Recoder framework.*

The approach and tool are successfully validated replacing a direct method call between a producer and a consumer component by communication via a buffer object.

1. Introduction

One general problem in software development is that we need to adapt the source code of our programs, because requirements change or because we want to compose independently developed components that do not completely fit each other. One main reason why components do not fit each other are *architectural mismatches* [6], i. e., the components make different assumptions on how to interact with each other.

The adaptations necessary to tailor a program to satisfy new requirements or to bridge mismatches in component composition essentially concern the interactions embodied in the program and among the components. This is due to the fact that nearly everything in a program represents an interaction, like input/output actions, variable accesses, calls to procedures, functions or methods, etc. In effect, composing components means to determine their interactions. Unfortunately, code dealing with an interaction is usually deeply buried in the component's code instead of being encapsulated in a separate first-class entity. Architecture systems [7] like Darwin [4], UniCon [24], Rapide [17] and Wright [1] try to address this problem by introducing *port* entities to represent interaction interfaces of components and *connector* entities to represent interactions among components. But these architecture systems do not deal with already existing code, i. e., they are unable to extract the architecture in terms of components and connectors of an existing system available as source code, for example. This also means, that these systems are unable to encapsulate interaction code deeply embedded in existing source code modules. These problems are known as *architecture reconstruction* and *aspect or feature extraction* problems [8, 14].

The above discussion shows that interaction behavior often is an implicit assumption that shows up as an aspect [16]. Consequently, adapting interactions involves invasively changing the corresponding components' source code. Sometimes, introducing wrappers as special connectors is successful, for example to mediate between different protocols [25]. But, introducing wrappers increases system complexity (amount of components and interactions) and decreases performance, since wrappers remain in the system as runtime entities, potentially imposing long chains of delegations. Moreover, introducing wrappers into a system often requires invasive changes itself.

In the sequel we therefore present our approach and our COMPASS tool to (semi-)automatically perform invasive as well as wrapping adaptations using program transformations realized as meta-programs, eliminating explicit connector elements in the final code whenever possible.

2. Approach

Our approach to consistently reconfigure interactions and automatically adapt the source code accordingly consists of five steps switching between the three levels of our architectural system model (cf. Figure 1).

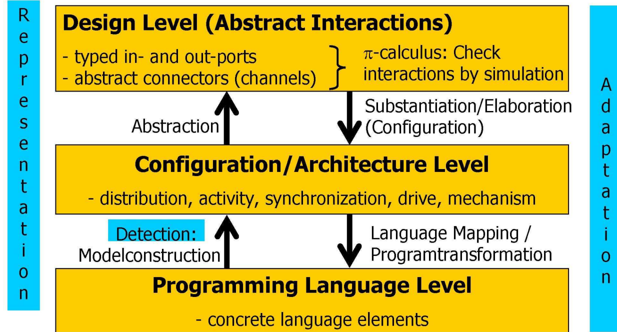

Figure 1. Interaction Configuration Levels.

The lowest level is the *programming language level*, that contains the concrete program to adapt. So the first step is the *detection phase (model construction)* that identifies components, their interactions and interaction patterns. The current version of our COMPASS tool can analyze Java sources based on the Recoder tool [19, 18]. This is done by parsing the sources, iterating the abstract syntax forest and using the definition table constructed by Recoder (name and type analysis). Interaction patterns (especially design patterns [5]) are identified combining static and dynamic analyses as described in our earlier papers [11, 10, 12, 13].

The result of this phase is a *configuration level model* constituting a program representation suited to manually specify interaction configurations. This level abstracts from the concrete programming language and represents all interactions explicitly as first-class entities combining the ideas of architecture systems (see above) and aspect-oriented programming [16] as well as hyperdimensional separation of concerns [21]. Figure 2 illustrates the elements of the configuration level and their links to corresponding source code elements.

The elements of the configuration level are components, aspect-oriented ports and aspect-oriented connectors.

Components are the basic building blocks of every system. Table 1 shows the mapping of Java syntax elements to COMPASS component model elements. Whenever the iterator of the detection and model construction phase encounters such a syntax node, it creates an instance of the corresponding COMPASS component model element. Of course, components may be hierarchically composed of further components including subsystems.

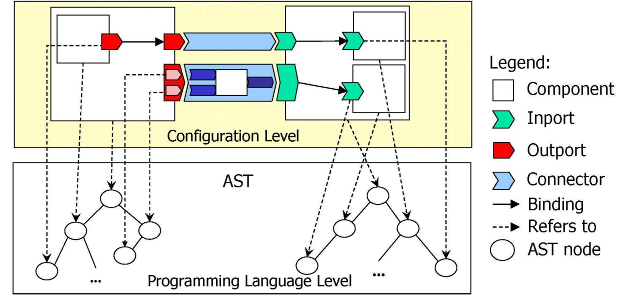

Figure 2. Configuration Level Elements, their Structure and their Links to Source Code.

Java element	COMPASS element
compilation unit	ModuleComponent
class declaration	ModuleComponent
method declaration	ProcedureComponent
field	FieldComponent
variable	VariableComponent

Table 1. Mapping of Java Syntax Elements to COMPASS Component Model Elements.

Aspect-oriented ports encapsulate the interaction points of components in an aspect- or hyperslice-like fashion and represent the interaction properties at these points. Interaction properties are for example the direction of the interaction (in or out), the type of the interaction (control or data), synchronization, drive (initiation of control flow), and interaction mechanism.

The model construction phase constructs a COMPASS port model element instance for every syntax node denoting data or control flow, like a method call for example.

Some ports of components are implicitly defined by the language semantics without an explicit syntactical representation in the programming language. Since our model aims at making all interactions explicit, we also create explicit port model element instances for those and associate them with their owning components. An example is the implicit first parameter (`this`) of a Java method, identifying the object to which the method belongs, i. e., the object which state has to be considered when referring to fields.

Aspect-oriented connectors that represent interactions by connecting ports. A connector also is a representation of a program transformation. Via the ports it connects, a connector has access to the interaction points buried in the components' code. It can thus substantiate or replace this code by the configured interaction code. Even connections established by private fields are considered, but the corresponding ports are marked as internal component ports.

These component, port and connector entities constitute an architectural model.

In the *(re-)configuration phase*, the developer reconfigures and adapts interactions by exchanging the port and connector entities on the configuration level. This triggers corresponding source code transformations (*transformation phase*) implemented as meta programs using the Recoder framework. Since connectors are architecture and design level instances by nature, our source code transformations eliminate them in the final code mapping them to potentially several constructs of the target language. Nevertheless, we retain the configuration level representation of the system as an architectural representation.

A program transformation consists of the two passes *analysis* and *transformation*. The *analysis pass* first collects the information necessary to perform the transformation by transitively identifying the ports and connectors affected by the transformation. These provide the relevant information already collected during the model construction phase. This especially comprises the source code elements to modify. Second, the analysis pass checks if the transformation can be executed, i. e., if the required information are available and the transformation is applicable in the given context. It might happen for example that a Java class is only available as byte code, so that we cannot transform its source code. Moreover, the configuration may define to connect incompatible ports using the wrong type of connector. The *transformation pass* performs the transformation without performing any further analyses. This is important, since a sub-transformation may have changed the underlying system already, so that the analysis now produces different or misleading results, although the whole complex transformation should be regarded as atomic. The whole transformation is encoded as a Java program, that modifies the abstract syntax forest of the sample program. The final target code is then produced using Recoder to pretty print the modified syntax forest.

The third architectural level of our model is the *abstract level* that abstracts from the concrete interaction properties represented on the configuration level. On the abstract level interaction consists of input and output actions on typed channels. This allows to bridge architectural mismatches or mismatches caused by certain implementation techniques.

An *abstraction phase* maps configuration level elements to abstract model elements, a *substantiation phase* conversely maps abstract model elements to configuration level elements by adding implementation specific interaction properties like synchronisation, drive and mechanism.

We have defined the semantics of all these levels formally using the π-calculus, i. e., every interaction is resolved to one or more input ($c(v)$) or output actions ($\bar{c}v$) on channels (c). The details are presented in [9].

3. Evaluation

We have validated our approach and our tool replacing a direct method call between a producer and a consumer component by communication via a buffer object. The source code produced as a result of applying the program transformation was syntactically and semantically correct: we did not obtain any compiler or runtime errors, the transformed program was executable and behaved as we wanted.

One might argue that the initial model construction phase is unnecessary, because every transformation performs its own analysis pass. This would even enable lazy evaluation instead of constructing the whole model in advance. Although this is technically true, this approach suffers from one enormous deficiency: to adapt a system or a set of components the developer needs an architectural model to identify system structure, especially the interactions and interaction patterns he potentially wants to reconfigure. The only admissible reason to dispense with the architectural model is when applying refactorings or when a model of the target system is already available.

4. Related Work

Most of the work in this area is concerned with general approaches to software design, decomposition, composition and evolution [16, 21, 2], program understanding and refactoring. Only few works [3, 22] specifically deal with adapting interactions.

Carriere et al. [3] generate program transformations to exchange communication primitives in C source code. First, they identify communication primitives using regular expressions and naming conventions. Then, they use the mapping defined in [15] to specify pre- and post-conditions to generate corresponding program transformations to be performed by the tool Refine/C [23]. These transformations then transform the abstract syntax tree of the program under consideration, automatically. This is demonstrated by example of exchanging a client-server communication via a Unix socket by a publish-subscribe-communication. This approach has the following shortcomings: the mapping of communication mechanisms is neither unique nor 1:1, thus leading to dead or superfluous code in the target system. Moreover, this includes the problem of deciding which target mechanism to use. In our approach this is done interactively or by providing a transformation strategy. Carriere et al. do not solve this problem. Furthermore, their approach does not detect complex interaction patterns. This is due to the fact that their analysis is based on naming conventions and regular expressions that are not powerful enough.

Pulvermüller et al. [22] encapsulate CORBA-specific communication code in aspects implemented using AspectJ. They further present how to achieve several differ-

ent system configurations by exchanging these aspects. The presented configurations are changing the location (local or remote) of a server component, and realizing a name server either as a CORBA Naming Service compliant name server or a file storing the names. This approach thus uses the general purpose AOP tool AspectJ to perform a few highly specialized transformations. But it does not define an interaction model, it does neither provide general support to adapt interactions nor to adapt interaction patterns. An analysis phase to detect interactions and provide a representation suitable to adapt them is completely missing.

Altogether, we do not know any approach that deals with the complete procedure of adapting interactions, as we did. The stepwise refinement of architectures has now become popular using the term *model driven architecture (MDA)* [20].

5. Conclusion

We have presented an approach that combines the ideas of architecture systems [7], aspect-oriented programming [16] and hyperdimensional-separation of concerns [21] to define *ports* that provide invasive access to interaction points of *components*, and *aspect-oriented connectors* as program transformations to consistently adapt interactions according to specified configurations. As an example, we have implemented and shortly discussed a program transformation that exchanges a direct method call by communication via buffer. To validate our approach, we have successfully applied this program transformation to a simple producer-consumer example system.

Our future work comprises to implement further transformations thus expanding our transformation library. Moreover, we need to deal with the problem of how to find the interactions to adapt: Given a problem specification or a catalog of known problems, which are the interactions to adapt? The detection of anti-patterns and the application of metrics to apply refactorings are promising in this domain, but need to be extended to non-meaning-preserving transformations, in the sense that desired new observable properties can be added to the target system.

References

[1] R. Allen and D. Garlan. Beyond definition/use: Architectural interconnection. In *ACM IDL Workshop*, volume 29(8). SIGPLAN Notices, 1994.

[2] U. Aßmann. *Invasive Software Composition*. Springer, 2002.

[3] S. J. Carriere, S. G. Woods, and R. Kazman. Software Architectural Transformation. In *WCRE 99*, Oct. 1999.

[4] Darwin. `http://www.doc.ac.ic.uk`, 2000.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 1995.

[7] D. Garlan and M. Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing, 1993.

[8] D. R. Harris, A. S. Yeh, and H. B. Reubenstein. Extracting Architectural Features From Source Code. *ASE*, 3:109–138, 1996.

[9] D. Heuzeroth. A Model for an Executable Software Architecture to deal with Evolution and Architectural Mismatches. Technical report, Universität Karlsruhe, 2004.

[10] D. Heuzeroth, G. Högström, T. Holl, and W. Löwe. Automatic Design Pattern Detection. In *IWPC*, May 2003.

[11] D. Heuzeroth, T. Holl, and W. Löwe. Combining Static and Dynamic Analyses to Detect Interaction Patterns. In *IDPT*, June 2002.

[12] D. Heuzeroth and W. Löwe. *Software-Visualization - From Theory to Practice, Edited by Kang Zhang*, chapter Understanding Architecture Through Structure and Behavior Visualization. Kluwer, 2003.

[13] D. Heuzeroth, W. Löwe, and S. Mandel. Generating Design Pattern Detectors from Pattern Specifications. In *18th ASE*. IEEE, 2003.

[14] R. Kazman and S. J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *ASE*, 6(2):107–138, Apr. 1999.

[15] R. Kazman, P. Clements, and L. Bass. Classifying Architectural Elements as a Foundation for Mechanism Matching. In *Proceedings of COMPSAC 97*, Aug. 1997.

[16] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented Programming. In *ECOOP'97*, pages 220–242. Springer, 1997.

[17] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE ToSE*, 21(4), 1995.

[18] A. Ludwig and D. Heuzeroth. Metaprogramming in the Large. In *GCSE, LNCS 2177*, pages 443–452, Oct. 2000.

[19] A. Ludwig, R. Neumann, and D. Heuzeroth. The RECODER project main page. `http://recoder.sourceforge.net`, 2002.

[20] OMG. MDA Guide Version 1.0.1. Technical report, OMG, 2003.

[21] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns in Hyperspace. Technical report, IBM T. J. Watson Research Center, Apr. 1999.

[22] E. Pulvermüller, H. Klaeren, and A. Speck. Aspects in distributed environments. In *GCSE'99*, Sept. 1999.

[23] `http://www.reasoning.com`, 2003.

[24] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE ToSE*, 21(4), 1995.

[25] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *OOPSLA'94*, Oct. 1994.