Modeling Component Based Embedded Systems Applications with Explicit Connectors in UML 2.0

Dietmar Schreiner and Karl M. Göschka
Vienna University of Technology
Institute of Information Systems, Distributed Systems Group
Argentinierstrasse 8 / 184-1, A-1040 Vienna
{d.schreiner,k.goeschka}@infosys.tuwien.ac.at

ABSTRACT

When building a system by connecting components, the connection itself, the connector, becomes a hot-spot of abstraction for any interaction. In contrary to most existing component models, we introduce explicit connectors as first class architectural entities. They materialize detailed contracts regarding composition, deployment and interaction and hence provide fine granular information on composed structures. Using explicit connectors results in custom-tailored and consequently light-weight middleware, as any interaction logic is contained within them. Modeling component architectures with explicit connectors allows the usage of off-the-shelf connector libraries. Thereby, developing a distributed component based application becomes less complex and more competitive due to reduced costs and increased reliability. We contribute by adopting a model driven development process for the use of explicit connectors by extending the syntax of UML 2.0 and defining a set of required model transformations.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures, Languages*; D.2.12 [**Software Engineering**]: Interoperability—*Distributed objects*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Design, Verification

Keywords

component, connector, UML, MDA, embedded systems

1. INTRODUCTION

In component based systems [3] the point of connection between components, namely the connector, represents an abstraction for any, even distributed or heterogeneous, component interaction. This rather complex process is typically transferred from application components into the component model's heavy weight implementation in order to make it transparent for the components themselves. Connectors within these component models are pure abstract entities representing middleware functionality and are called implicit connectors, as their logic is implicitly contained within the component model's middleware.

In embedded systems the application of heavy-weight middleware is often disadvantageous due to the system's limited resources. Nevertheless, it is a good idea to keep the complex and error-prone interaction logic separated, if possible hidden, from the application components. This can be achieved by introducing coherent and explicit connectors and their contracts in a component model. In addition, by using explicit connectors, detailed requirements and provisions regarding component composition and interaction become visible. These emerging contracts allow a more precise model level validation of component architectures.

2. THE EXPLICIT CONNECTOR

An explicit connector is an architectural entity used to represent component composition and interaction in distributed heterogeneous systems. Although explicit connectors have a great similarity to components, they differ in many aspects: In contrary to components, a connector changes its materialization during its life-cycle and is subject to heavy transformation: (i) In platform independent models *(PIM)* the explicit connector is an abstract representation of a component interconnection, specifying the type of interaction. (ii) In platform specific models *(PSM)* the explicit connector is transformed into a set of distributed fragments, which in total implement the functionality of that specific explicit connector. Connector fragments are deployed along with their associated components and are hierarchically composed structures themselves. All connectors remaining in platform specific models are implicit connectors, typically procedure calls. (iii) At deployment- and finally at run-time connectors are no longer visible. True components, representing the explicit connectors' functionality, are executed.

3. EXTENDING UML 2.0

To model component based applications using explicit connectors in UML 2.0, the UML syntax has to be extended. UML 2.0 specifies two types of connectors: (i) the assembly connector and (ii) the delegation connector. When talking

SAC '07, March 11-15, 2007 Seoul, Korea

about connectors within this paper, we refer to assembly connectors and their extensions.

Every connector is made up of two parts: The two connected interfaces of the composed components. When taking a closer look at the interfaces' role and properties it is obvious that one component interface is active while the second interface is reactive. For example a procedure call connector is made up by connecting a required and a provided interface. The *required interface* is the active part of the connector, as the component requiring a procedure via this interface is the one calling it. The *provided interface* is the reactive part. The provided procedure is inactive and gets activated only when called. In broadcast connectors, the active interface is the data emitter or *sending interface* while the reactive part is the *receiving interface*. A detailed table of connectors with their parts, the active and reactive interfaces, and their notation used to extend UML 2.0 component diagrams is given in Figure 1. The notation of the *Procedure Call Connector*, labeled with *, overrides the default UML 2.0 connector. The notation of the *Data Broadcast Connector*, labeled with ** is chosen in accordance to the *AUTOSAR* component specification [1].

Connector Type	Active Interface		Reactive Interface		Connector Notation
	Name	Notation	Name	Notation	
Procedure Call	Client (Required IF)		Server (Provided IF)		*
Data Broadcast	Sender		Receiver		**
Black Board	Accessor		Provider		

Figure 1: Notation of Explicit Connectors

4. MODEL DRIVEN DEVELOPMENT

4.1 Composition and Distribution

When building an application, its platform independent component architecture has to be specified by assembling all required components within a component diagram. Components and connectors are associated with related contracts to enable model level validation. At this stage of development explicit connectors are used to model the type of component interaction on a very abstract level. A more detailed description of contracts and their transformation as well as the calculation of emerging compositional contracts is provided in [4]. In addition, the components' distribution has to be specified in a deployment diagram. As explicit connectors are abstract entities at this stage of development, they are not visible in the platform independent deployment specification.

4.2 Transformation

The heart of our proposed software engineering approach is the automated model transformation of the specified platform independent models (see Section 4.1 into platform specific models containing real entities providing the functionality of all explicit connectors. These entities turn out to be composed structures, that map to connector fragments. They are associated with interface- and component contracts that emerge from the transformation and specify additional requirements and provisions. The transformation itself is done in three main steps.

4.2.1 Step 1: Connector Selection

By specifying the components' deployment scenario all information about the target platforms and available communication channels for component interaction becomes available. Based on that information, components implementing the connector fragments' functionality have to be selected. As mentioned before the abstract connector is made up of two fragments, each attached to one of the two connected components. As result of this transformation step, the abstract explicit connector is replaced by real components from connector libraries.

4.2.2 Step 2: Adapter Generation

Connector fragments are standardized components from connector libraries, thus their default interfaces do not match the interfaces of the connected application components. The next phase of the model transformation is therefore the generation of interface adapters. These adapters are again components that translate one interface to another.

4.2.3 Step 3: Recomposition and Deployment

Finally all components, the application components, the connector fragments and the generated adapter components have to be recomposed to form the transformed platform specific model of the developed application. This new model contains a large amount of additional platform specific information represented in additional component- and interface-contracts.

5. ADDED VALUE OF OUR APPROACH

The benefits gained by the presented software engineering process are manifold. (i) Application components don't have to deal with interaction issues due to explicit connectors from off-the-shelf connector libraries. (ii) Emerging contracts from the connector transformation provide detailed distribution related information for model level validation. (iii) The set of all explicit connector fragments deployed on the same ECU represents the application's ECU specific smallest possible middleware.

6. ACKNOWLEDGEMENTS

This work has been partially funded by the FIT-IT [embedded systems initiative of the Austrian Federal Ministry of Transport, Innovation, and Technology] and managed by Eutema and the Austrian Research Agency FFG within project COMPASS [2] under contract 809444.

7. REFERENCES

[1] AUTOSAR. *Automotive Open System Architecture*. http://www.autosar.org/.

[2] COMPASS. *Component Based Automotive System Software*. http://www.infosys.tuwien.ac.at/compass.

[3] B. Meyer. The grand challenge of trusted components. In *ICSE*, pages 660–667, 2003.

[4] D. Schreiner and K. M. Göschka. Explicit connectors in component based software engineering for distributed embedded systems. In *SOFSEM07*. Springer, to appear.