Synthesizing Communication Middleware from Explicit Connectors in Component Based Distributed Architectures

Dietmar Schreiner[1,2] and Karl M. Göschka[1]

[1] Vienna University of Technology
Institute of Information Systems, Distributed Systems Group
Argentinierstrasse 8 / 184-1, A-1040 Vienna, Austria
{d.schreiner,k.goeschka}@infosys.tuwien.ac.at
[2] University of Applied Sciences Technikum Vienna
Department of Embedded Systems
Höchstädtplatz 5, A-1200 Vienna, Austria

Abstract. In component based software engineering, an application is build by composing trusted and reusable units of execution, the components. A composition is formed by connecting the components' related interfaces. The point of connection, namely the connector, is an abstract representation of their interaction. Most component models' implementations rely on extensive middleware, which handles component interaction and hides matters of heterogeneity and distribution from the application components. In resource constrained embedded systems this middleware and its resource demands are a key factor for the acceptance and usability of component based software. By addressing connectors as first class architectural entities at model level, all application logic related to interaction can be located within them. Therefore, the set of all explicit connectors of a component architecture denotes the exact requirements of that application's communication and interaction needs. We contribute by demonstrating how to use explicit connectors in model driven development to synthesize a custom tailored, component based communication middleware. This synthesis is achieved by model transformations and optimizations using prefabricated basic building blocks for communication primitives.

1 Introduction

Driven by market demands, the application of embedded systems experienced a significant upturn over the last years. A wide variety of new fields of application as well as more demanding requirements in established ones lead to a tremendous boost in complexity of embedded systems software. Today's embedded applications are no longer simple programs executed on one single electronic control unit (ECU). In fact, they are heterogeneous software systems in distributed and often safety or mission critical environments and hence have to be small, efficient but also extremely reliable.

M. Lumpe and W. Vanderperren (Eds.): SC 2007, LNCS 4829, pp. 160–167, 2007.
© Springer-Verlag Berlin Heidelberg 2007

1.1 State of the Art

A well accepted approach in developing cost-efficient and sound embedded systems software is that of component based software engineering (CBSE): applications are built by assembling small, well defined and trusted building blocks, so called components.

In accordance to the work of [1,2,3,4] a component is a (i) trusted architectural element, an element of execution, representing (ii) software or hardware functionality, with a (iii) well defined usage description. It conforms to a (iv) component model and can be independently composed and deployed without modification according to the model's composition standard. At run-time components interact through their provided and required interfaces conforming to the component model's interaction standard. Therefore, components are reusable and exchangeable.

The process of interaction may become rather complex especially in distributed heterogeneous systems. As it is good practice to keep application components simple and focused on their primary purpose, any program code related to interaction handling has to be separated from the application component's implementation. This is typically done by introducing communication middleware handling all types of interaction in a transparent way. Interacting application components utilize that middleware and therefore face their distribution and deployment scenario as configuration issue only.

1.2 Contribution

Using a general purpose communication middleware seems to be a great advantage at first glance, but turns out to be rather cumbersome in resource constrained systems. As such middleware has to cope with all possible types of interaction, implementations tend to be rather heavy-weight pieces of monolithic software. Since resource consumption is a key factor in embedded systems software development, we provide an overview on how application specific communication middleware can be synthesized from software models containing explicit connectors within a model driven development process. This custom tailored middleware exactly covers the application's communication needs and therefore helps in building a light-weight component middleware for embedded systems. In addition the proposed approach leads to component based middleware, so available mechanisms and tools from the domain of CBSE (e.g. model verification techniques) can be applied to the middleware itself.

1.3 Overview

Section 2 describes the types of connectors in component models. It gives a detailed view on explicit connectors, as they are used to generate communication middleware in our approach. The general structure of component middleware feasible for embedded systems is described in Section 3. Section 4 finally gives an overview on how communication middleware can be synthesized in model

driven development by transforming and optimizing application models, utilizing prefabricated basic communication primitives.

2 Implicit and Explicit Connectors

Two components may interact at runtime if their related *provided-* and *required interfaces* are validly associated at composition time. This association, namely the connector, is an abstract representation of any interaction occurring between the connected components. As mentioned before, in most component models the process of interaction is covered within middleware, therefore we consider these connectors to be implicit.

An explicit connector is an architectural entity, that is used to represent component composition and interaction and owns its own implementation of interaction operators. Therefore an explicit connector encapsulates all communication logic for one specific type of interaction. In addition, it specifies properties of the connected components' interaction and provides contracts regarding communication channels and resource requirements.

Modeling component based applications using explicit connectors in UML 2.0 requires the UML component syntax to be extended. UML 2.0 specifies two types of connectors: (i) the assembly connector and (ii) the delegation connector. When talking about connectors within this paper, we refer to assembly connectors and their extensions.

To keep explicit connectors small in size, they have to be highly specialized in type and target platform.

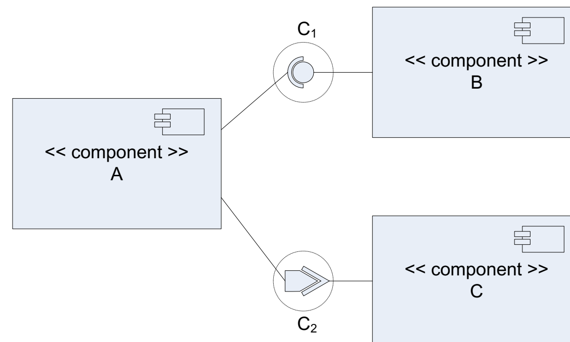

Fig. 1. Client-Server and Sender-Receiver Connector

Explicit connectors fall into two main classes:

Client-Server Connectors: A component providing a service is called server, a component using that service is called client. The client-server connector connects components of this type. Typical client-server connectors are those

connecting procedural interfaces. Figure 1 shows a client-server connector labeled C_1 where component A is the client and component B is the server.

Sender-Receiver Connectors: These connectors provide means of non-blocking, one-to-many and many-to-one data distribution. Sender-receiver connectors typically implement a "last-is-best" semantic—only the last received data value is valid and accessible—and are used to connect components emitting and collecting data. Figure 1 also shows a sender-receiver connector labeled C_2 where component A is the sender and component C is the receiver.

A detailed classification of explicit connectors for the domain of automotive embedded systems according to the *AUTOSAR* [5] standard was provided within the project *COMPASS* [6], but is out of the scope of this paper.

Although explicit connectors have a great similarity to components, they differ in many aspects: In contrary to components, a connector changes its appearance during its life-cycle due to model transformations. (i) In platform independent models the explicit connector is an abstract representation of component interconnection, specifying properties of the interaction type. (ii) In platform specific models 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 themselves composed structures made up of basic components. Any connectors that remain at the platform specific level after applying all transformations are implicit connectors, typically local procedure calls. (iii) At deployment- and finally at run-time the explicit connector is no longer visible. True components, representing the explicit connectors functionality, are deployed and executed.

Figure 2 depicts a connector fragment of connector C_2 from Figure 1 in a platform specific model. It consists of a (generated) interface adapter

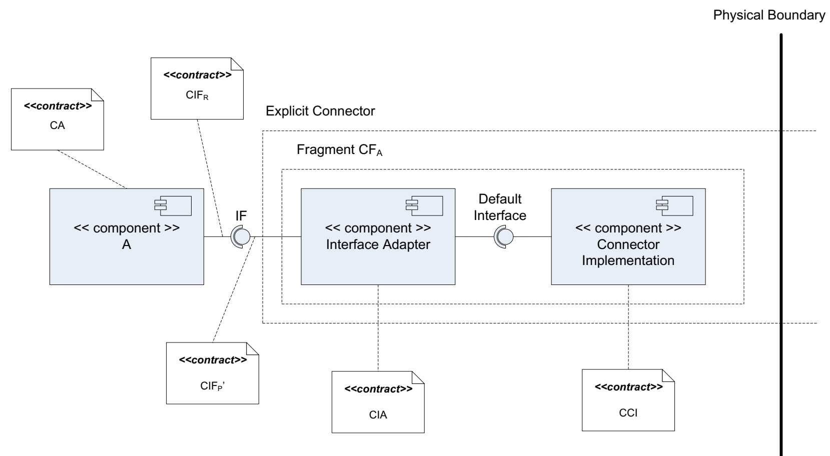

Fig. 2. Connector Fragment

component and a generic sender component. The two client-server connectors left over in this model are implicit local procedure calls and do not have to be transformed any further. In addition, Figure 2 shows various contracts associated with components and interfaces. These contracts can be used for a more detailed model verification of the constructed component architecture [7,8,9].

3 Middleware

As described in Section 1, the process of interaction in component architectures is typically handled by middleware. Middleware is an additional software layer, that is located between an application and the operating system and its communication stack. Component middleware additionally manages the life-cycle of components, handles their interaction, no matter if local or distributed, and provides infrastructural services for the components.

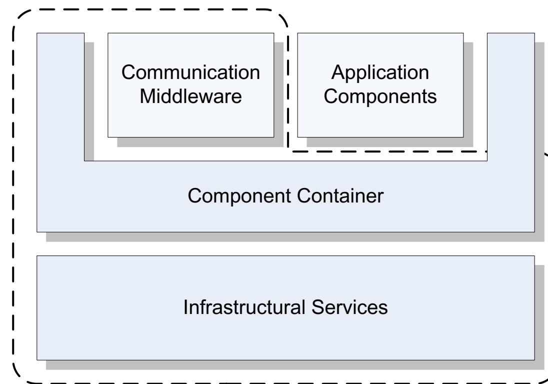

Fig. 3. Component Middleware

Figure 3 shows a simplified version of component middleware (encircled by the dashed line) that meets the requirements of distributed embedded systems applications. In safety critical applications of that domain, components have to be bound statically, instantiation occurs only at initialization time and communication channels are statically predefined. The component middleware therefore consists only of a (i) *component container* that hosts all components and manages their life-cycle and instantiation and of (ii) *infrastructural services* that are required by the container but may also be provided for the components. All components residing within the container may interact locally. To enable remote interaction with a distributed system, the component middleware finally includes (iii) *communication middleware*.

As one can see, we located the communication middleware inside the component container. This is because we synthesize communication middleware from

the application model and basic building blocks, that are simple components themselves. This process of synthesis is presented within the next section.

4 Middleware Synthesis

Explicit connectors as introduced in Section 2 encapsulate the implementation of the specific interaction process of the connectors' type. Application components interact via their interfaces, that are linked by explicit connectors. No other means of interaction are allowed within component models. This consequently implies, that the set of all explicit connectors of a component architecture covers the architecture's communication requirements. By transforming the application's models, explicit connectors are transformed into connector fragments, that themselves are transformed into various components like senders, receivers, protocol handlers or interface adapters. By eliminating redundant building blocks within the transformed architecture, the total set of required communication related components, the component middleware, can be calculated and deployed.

We are going to demonstrate the synthesis of communication middleware with a simplified example. This is no real world application but it will show the basic idea of our approach without exceeding this paper's page limit.

4.1 Application Specification

As first step in creating an application with a model driven process we define the component architecture in a platform independent model by assembling all application components within a UML 2.0 component diagram. All *required interfaces* are connected to the corresponding *provided interfaces* by specifying explicit connectors.

Figure 1 depicts the platform independent model of our demonstrator application. Component A requires services provided by component B by a procedural interface and provides data to component C by a sender-receiver interface.

The second step in developing our application is to specify the deployment scenario. For our example we deploy the application components on two distinct ECUs. Component A will be deployed on the first ECU while components B and C will be deployed on the second one. Note that explicit connectors at this stage of development are considered to be abstract entities, consisting of fragments, and therefore must not be included in the deployment specification.

4.2 Connector Transformation

By defining the component architecture and its deployment, all information required to transform the connectors into components becomes available. The so called *connector transformation* selects the proper connector implementations (e.g. a remote procedure call connector) from a connector library. It connects identified connector fragments to the application components. To do so, it generates interface adapter components to match all interfaces and modifies the

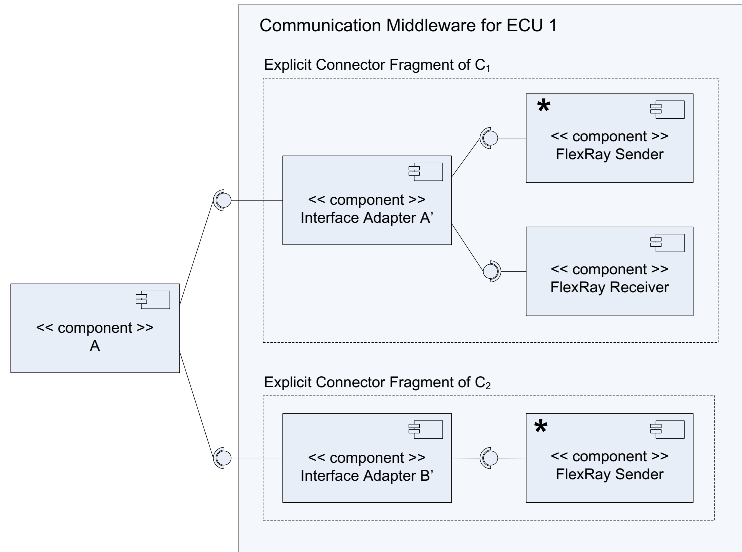

Fig. 4. Example Transformation for ECU 1

deployment specification to cover the inserted components, that in total represent the connectors' functionality.

Figure 4 depicts the result of the *connector transformation* for the example application part deployed on ECU 1. One can see, that the application component A now is connected to two connector fragments, fragment C_1 for a procedure call connector to component B and fragment C_2 for a data emitter connector to component C. Generated interface adapters map the application component's interface to the generic building blocks from the connector library. The used procedure call connector is a remote one, as the connected application components do not reside within the same address space—remember that we deployed them on two distinct ECUs)—and uses a receiver to get results back from component B.

All five components within the outer box are not application components and deal with interaction related issues only. Together, they assemble the custom tailored communication middleware for ECU 1.

4.3 Architectural Optimization

In a final step, the generated component architectures have to be optimized to eliminate redundant elements and meet additional system constraints like contractually specified uniqueness of specific components (e.g. singletons).

In our example, the sender components, both labeled with (*), redundantly exist within our middleware. To optimize the middleware's size these redundancy can be eliminated by sharing the sender between both connector fragments.

4.4 Summary

Component based software engineering is a well established engineering paradigm for distributed embedded systems. However, state-of-the-art component models often rely on heavy-weight component- and communication middleware to keep application components small and simple. The middleware's resource usage is a crucial factor in resource constrained systems. We provided an overview on how to synthesize the communication part of a component middleware from application models in order to keep it small. To enable this approach, we introduced explicit connectors as first class architectural entities at model level. In addition, we described how their implementation is performed by composing basic building blocks, stored within a connector library. By following our approach, the set of all explicit connectors within the application's platform independent model will be transformed into a custom tailored, light-weight communication middleware for each deployment node. Moreover, methods of verification for component architectures can be applied not only to the application but also to parts of its middleware.

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 [6] under contract 809444.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (1998)
2. Meyer, B.: The grand challenge of trusted components. In: ICSE 2003, pp. 660–667 (2003)
3. Heineman, G.T., Councill, W.T. (eds.): Component-Based Software Engineering. Addison-Wesley, Reading (2001)
4. Nierstrasz, O., Tsichritzis, D. (eds.): Object-Oriented Software Composition. Object-Oriented Series. Prentice-Hall, Englewood Cliffs (1995)
5. AUTOSAR (Automotive Open System Architecture), `http://www.autosar.org/`
6. COMPASS (Component Based Automotive System Software), `http://www.infosys.tuwien.ac.at/compass.`
7. Schreiner, D., Göschka, K.M.: Explicit connectors in component based software engineering for distributed embedded systems. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 923–934. Springer, Heidelberg (2007)
8. Reussner, R.H., Schmidt, H.W.: Using parameterised contracts to predict properties of component based software architectures. In: Ivica Crnkovic, S.L., Stafford, J. (eds.) Workshop on Component-based Software Engineering Proceedings (2002)
9. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Software Eng. 28(11), 1056–1076 (2002)