Explicit Connectors in Component Based Software Engineering for Distributed Embedded Systems

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. The increasing complexity of today's embedded systems applications imposes the requirements and constraints of distributed, heterogeneous subsystem interaction to software engineers. These requirements are well met by the component based software engineering paradigm: complex software is decomposed into coherent, interacting units of execution, the so called components. Connectors are a commonly used abstraction to model the interaction between them. We propose to use explicit connectors when building distributed embedded systems applications. Explicit connectors encapsulate the logic of distributed interaction, hence they provide well defined contracts regarding properties of inter-component communication. Our approach allows model level validation of component composition and interaction incorporating communication related constraints beyond simple interface matching. In addition, by using explicit connectors, the complexity of application components is reduced without the need for any heavy weight middleware.

1 Introduction

Currently embedded applications are no longer simple programs executed on single electronic control units (ECUs). In fact, nowadays embedded systems applications are heterogeneous software systems, deployed on a wide variety of hardware platforms and communication subsystems. In addition embedded systems applications are often used in safety or mission critical environments.

This all lead to a dramatic increase of software complexity and consequently to an increase of erroneously deployed software. To overcome that problem and to reduce the overall costs for embedded systems applications, various paradigms from the classical software engineering process have been adopted to the needs of the embedded systems domain. Adoption becomes necessary due to the limited resources in embedded systems, which would otherwise render many useful concepts from the classic software engineering domain unusable. The limitations range from that of processing power over available memory and network-bandwidth up to safety and real-time issues. In general, embedded applications have to be small, efficient and extremely reliable.

1.1 Background

A widely accepted and adopted software engineering paradigm within the embedded systems domain is that of component based software engineering (CBSE). The key concept behind CBSE is to construct an application by composing small, simple units of execution - the components. Components are specified by their interfaces, contracts [8], and their accordance to a specific component model. As components provide means of exchangeability and reusability, implicit context dependencies are strictly prohibited.

When building a system by composition, so by connecting components, the point of connection, the connector, becomes a hot-spot of abstraction for any interaction. In many component systems like Enterprise Java Beans [18], the Corba Component Model [14], or DCOM [10], the rather complex process of distributed, heterogeneous interaction is transferred from the individual components into the component model's heavy weight middleware implementation in order to make it transparent for the components themselves. In these component models connectors are abstract model level representations of component interaction and are typically not associated with any contractual properties beyond function signatures within interface specifications.

In embedded systems the application of a heavy-weight middleware is often disadvantageous due to the systems' 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. In our approach this is achieved by introducing coherent and explicit connectors and associated contracts in the component model. In addition, by using explicit connectors, more precise requirements and provisions regarding the components' interaction become visible. These emerging contracts allow a detailed computation of requirements and may be used for model level validation of component composition.

1.2 Contribution

We demonstrate how to use explicit connectors at model level when building component based applications for distributed embedded systems. The advantages gained by this approach are threefold: (i) By encapsulating the interaction and communication logic within connectors, the complexity of application components is reduced. Application components become smaller in size and complexity but more reusable and reliable. (ii) Connectors can be provided off-the-shelve (OTS) by communication subsystem suppliers. This will also reduce the development costs of a distributed embedded systems application and increase its reliability. (iii) Explicit connectors home all interaction and communication logic. Hence they provide an additional set of contracts that emerge from the component architecture and the deployment specification. We show how to use these emerging contracts to improve the validation of component based applications at model level.

2 Components, Connectors and Contracts

In accordance to the work of [19, 9, 7, 11] we define a component as 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 deployed and composed without modification according to a composition standard.

An interface is a set of exposed services through which components interact. A provided interface exposes a components functionality for usage by other components while a required interface specifies the need of functionality of other components. As interfaces are the only points of component interaction, a component has to provide at least one interface, but may own multiple, distinct ones, so called facets. Interfaces specify the dependencies between the services provided by the component and the services required to fulfill the component's task.

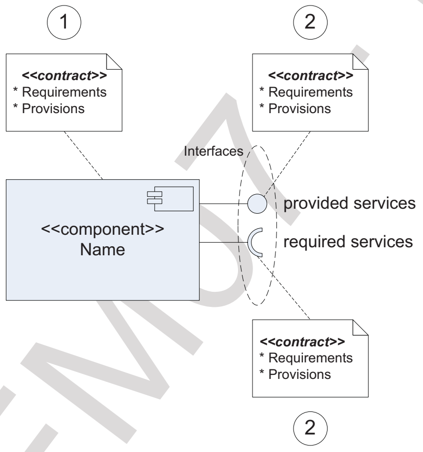

Fig. 1: UML 2.0 notation of a component

Figure 1 shows the notation of a component and its interfaces complying with the UML 2.0 Superstructure specification [13]. The UML 2.0 notation will be used for most figures within this paper.

To strengthen the reliability and predictability of component based applications, guarantees about the behavior of application elements are formalized in contracts [8, 15]. Contracts specify requirements and provisions of associated elements. In general a contract consists of two obligations:

1. The client, requiring functionality from another element, has to satisfy the preconditions of the provider.

2. The provider, that is the supplier of the required functionality, has to fulfill its postcondition, if the client's precondition is met.

We distinguish four types of contracts:

1. Component-contracts are associated with components and their instances. Typical component contracts deal with resource requirements or deployment restrictions.
2. Interface-contracts specify services and properties of the components' interfaces like function signatures or temporal properties for interface invocations.
3. Connector-contracts are associated to connectors and deal with constraints related to the used communication channels.
4. Platform-contracts specify properties of platform elements like ECUs or bus systems regarding provided memory or timing information.

In Figure 1 contracts for the component and each of its interfaces are specified. The one labeled with 1, is a component-contract, specifying requirements and provisions of the component itself. The others, labeled with 2, are interface-contracts, specifying requirements and provisions for interaction on a specific interface.

To build a valid application in CBSE, components are assembled to form a composed entity with a new behavior. To assemble means associating related provided and required interfaces. It is obvious that related interfaces have to be of the same type, so provide compatible interface-contracts. The connection between two components is called connector.

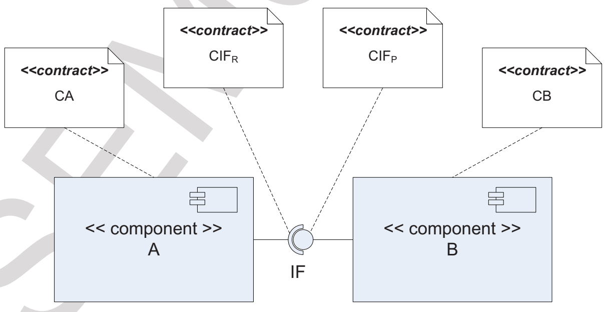

Fig. 2: Simple composition in UML 2.0

An example composition is depicted in Figure 2: two components *A* and *B* are connected to form a composed structure. *A* requires functionality provided by *B*. Therefor *B* exports that functionality by a provided interface *IF* denoted by a ball, *A* exports the requirement by a required interface *IF* denoted by a socket. As the type of *A*'s required interface is the same as *B*'s provided interface, the composition is legal. In addition four very basic contracts are specified within this figure: *CA* and *CB* are component-contracts specifying the component's resource requirements. CIF_R and CIF_P are interface-contracts for the required and provided interface of *A* and *B*. Figure 3 shows the interface contracts, that are very simple ones but are sufficient for demonstration purpose: both contracts refer to the same interface (*id=0*). Both interfaces are of the same type (*type="API")* and contain the same service (*id="exampleService"*) with an identical signature. However, the contracts differ in the worst-case-execution-time (WCET) property (*wcet*) of the service. As one can easily see, the provided WCET is less than the requirements, so the depicted composition seems to be valid.

```
<contract type="RI" id="CIFR">
  <interface type="API" id="0">
    <service id="exampleService">
      <param idx="0" type="void"/>
      <result type="void"/>
      <wcet t="0.05s"/>
    </service>
  </interface>
</contract>
```

(a) CIF_R

```
<contract type="PI" id="CIFP">
  <interface type="API" id="0">
    <service id="exampleService">
      <param idx="0" type="void"/>
      <result type="void"/>
      <wcet t="0.01s"/>
    </service>
  </interface>
</contract>
```

(b) CIF_P

Fig. 3: Interface contracts

Connectors as introduced in [17] represent first class architectural entities embodying component interaction. With increasing application complexity and distribution, connectors become a key factor in the whole development process. They encapsulate and therefore abstract the transfer of control and data among components. In many component models connectors are not mentioned explicitly as the implementation of the connectors' functionality resides within component middleware and is transparent for the application components.

In this paper connectors are considered to be explicit and thereby are granted a component equivalent status. This is mandatory as resource limited embedded systems typically lack complex component middleware or even real operating systems. Although explicit connectors look very similar to components, there exist two major differences:

1. As pointed out in [2], connectors are physically fragmented. When deploying two connected components on two different ECUs, the connector between the application components has to be split into two separate fragments, each deployed, and therefor colocated, with the related application component.
2. A connector's life-cycle starts after the specification of the components' deployment. Before the specification of the application's deployment schema,

connectors are abstract entities within architectural models. After specifying the physical component location, the available communication channels between the components are defined. This information is required to transform the abstract model entities into real, deployable connector fragments.

3 Using Explicit Connectors

In this section we demonstrate the usage of explicit connectors when building a component based application. We do this with a very simple application consisting of two connected components. This is of course no real-world application, but it is sufficient to demonstrate our approach. A more realistic application - an automotive, speed-aware lock control - has been implemented in the scope of project COMPASS [6] to proof our concept.

3.1 Component Architecture

The first step in developing a component based application is to define the application's architecture. We do this by specifying a UML 2.0 component diagram. Figure 2 depicts such a diagram. In addition to the composition of the components' interfaces, the connector's base type can be specified.

The base type of a connector can be derived by the connector's communication style. We identified several typical communication styles like procedure call, data broadcast, blackboard access or data stream and extended the UML 2.0 syntax for composition diagrams with symbols for explicit connectors. Example connector symbols are shown in Figure 4. A detailed classification of connector types is out scope of this paper, but is subject to ongoing research.

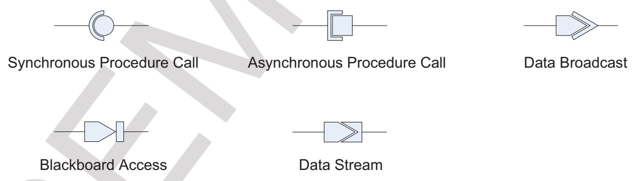

Fig. 4: Connector type symbols

For our example we use the application specified in the component diagram given in Figure 2. The application consists of two components *A* and *B*, one synchronous procedure call connector for interface *IF* and four associated contracts.

3.2 Deployment Specification

The next step in developing the application is to specify the deployment schema. Figure 5 provides a UML 2.0 deployment diagram: The sample application is distributed over two ECUs, *ECU1* and *ECU2*, that are connected by a physical bus

BUS. The ECUs and the bus are associated with platform contracts containing information about provided memory for each ECU or propagation delays on the bus.

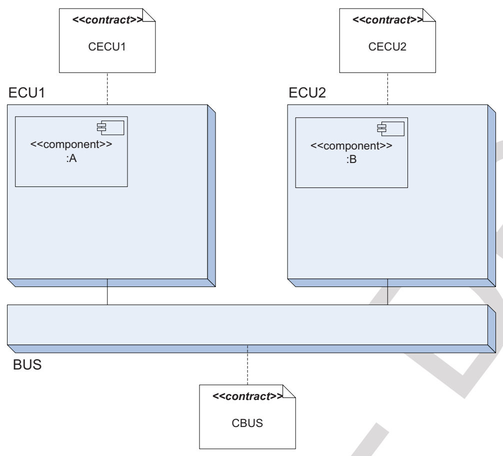

Fig. 5: Deployment schema

3.3 Transformation

By using the deployment specification, the component architecture can be transformed into a new one, containing concrete explicit connectors, to be more precise: connector fragments. In addition an adopted deployment scheme is generated too.

In our example the components A and B are located on different ECUs, that are connected by the bus *BUS*. The communication style of the connector is *synchronous procedure call*. Therefor the connector consists of two fragments, which have to be selected from the connector library of the used bus system and ECU. The transformed composition diagram of the application is denoted in Figure 6, the transformed deployment schema in Figure 7. Figure 6 shows that four additional contracts become available within the application model:

– The contracts CCF_A and CCF_B are connector-contracts. These contracts contain requirements of the connector fragments, similar to component-contracts.
– CIF'_P and CIF'_R are interface-contracts associated with the fragments' interfaces. The connector's interface-contract CIF'_P is calculated by extending A's interface-contract CIF_P with information provided by the connector-contracts CCF_A, CCF_B and the platform-contract of the bus *CBUS*.

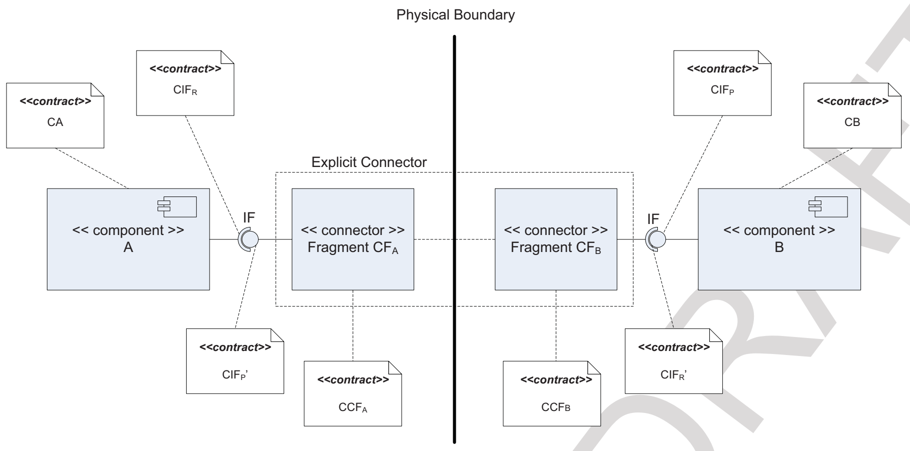

Fig. 6: Transformed composition diagram

These emerging contracts become extremely valuable when validating the constructed application model.

To enable the usage of a general connector library, an additional interface adaptor is required between the components and the general connector fragments. This adaptor is generated as part of the applied transformation process and is not shown in our example as it can be treated like an application component.

3.4 Validation

Finally the transformed model of the application can be validated. All available contracts have to be checked. To show the advantage of our approach, we will choose platform- and connector-contracts that will lead to an invalid application, although the constructed model seemed to be a valid composition as demonstrated in Section 2.

First all component- and connector-contracts have to be checked against the platform-contracts as specified in the transformed deployment diagram. In our example we assume, that the total of used resources on each ECU is less then the provided amount and that no hardware restrictions are violated by the components and the connector fragments. So the first validation check is passed successfully.

Next the interface-contracts have to be checked. We have to match the interface-contract CIF_R of component A with the emerging interface-contract CIF'_P of the connector.

To do so, CIF'_P has to be calculated: We have to create a new contract based on component B's interface-contract CIF_P using information provided by the

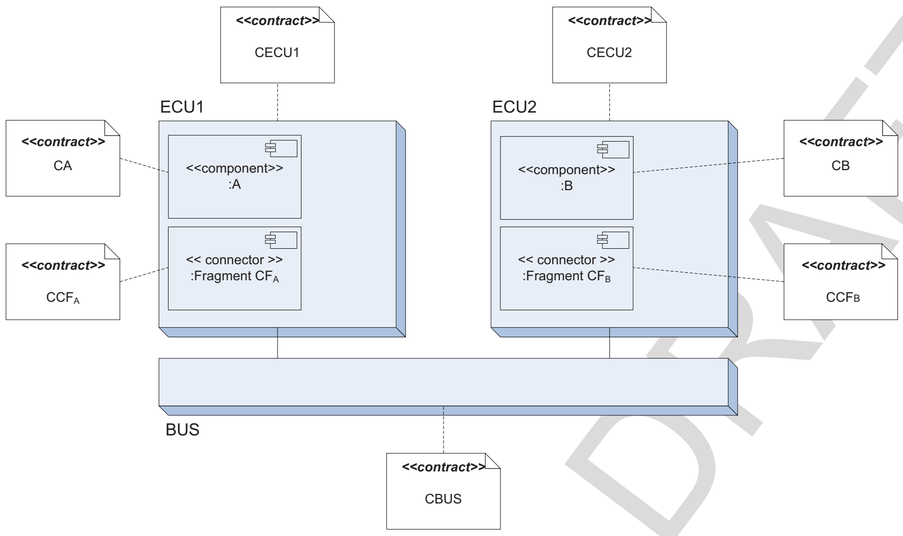

Fig. 7: Transformed deployment diagram

```
<contract type="P" id="CBUS">
  <bus id="0">
    <buscycle_length t="0.1"/>
    <slot_length t="0.05"/>
  </bus>
</contract>
```

(a) $CBUS$

```
<contract type="C" id="CCF?">
  <connector type="RPC">
    <response time="1*"/>
    <WCET t="0.01"/>
  </connector>
</contract>
```

(b) CCF_B,CCF_A

Fig. 8: Platform- and connector-contracts

connector-contracts and the platform-contracts of the communication subsystem. In Figure 8 the platform-contract of the bus and the connector-contracts for the fragments are specified. The connector-contracts are identical, so we just show one to save space. The connector fragments add additional execution time of 0.01 seconds each to the WCET in contract CIF'_P. As the connector type is *synchronous procedure call*, invoking a remote procedure requires a confirmation response containing the result. This implies that we again have to increase WCET in contract CIF'_P by the systems response time. That is calculated by multiplying the connectors *response time* with the *buscycle_length* of the bus. The so calculated emerging contract is given in Figure 9.

As one can see, the provided WCET of 0.13 seconds is higher than the required WCET of 0.05 seconds. Our sample application turned out to be invalid under the specified platform and deployment schema.

```
<contract type="PI" id="CIFP'">
  <interface type="API" id="0">
    <service id="exampleService">
      <param idx="0" type="void"/>
      <result type="void"/>
      <wcet t="0.13s"/>
    </service>
  </interface>
</contract>
```

Fig. 9: Calculated interface contract CIF'_P

Similar calculations can be applied to different functional and non-functional properties specified within the model's contracts.

4 Conclusion and Future Work

In our paper we demonstrated how to use explicit connectors when building component based applications. Explicit connectors help to reduce the complexity of application components in absence of a component middleware. Moreover, the set of all deployed connector fragments within one ECU can be seen as the custom tailored middleware for that specific ECU.

When using explicit connectors, additional contracts emerge from model transformation, using deployment information to specify the available communication channels. This leads to a more precise model level validation of component interaction in composed software architectures.

Our ongoing research deals with the identification and classification of connector types in automotive embedded systems applications and with the model level validation of looped composed component architectures.

5 Related Work

To adopt the *CORBA Component Model (CCM)* [14] to embedded software design connectors are integrated into CCM in *The CORBA Connector Model* [16]. Here, connectors are used to mediate interaction between distributed CORBA components and therefore are limited to CORBA specific interaction and communication styles.

Connectors in general are extensively examined within work [2, 4, 1] related to the project *SOFA - Software Appliances* [5]. *SOFA* defines a component model, providing hierarchically nested components and connectors as first class architectural entities. The internal structure of connectors is analyzed in *Communication Style Driven Connector Configurations* [4] aiming at automatic component composition. *SOFA* defines three types of basic connectors: (i) Procedure Call, (ii)

Event Delivery and (iii) Data Stream. In addition user-defined connectors can be specified. However, our research is focused on software connectors for embedded systems and therefore deals with more hardware and system related issues like e.g. resource usage of connectors.

Another project dealing with component based software engineering is *FRACTAL* [12]. *FRACTAL* defines a component model, that also contains connectors. A binding is defined to be a communication path between component interfaces. Bindings are classified to be (i) primitive or (ii) composite. A primitive binding binds one client interface and one server interface, in the same address space. A composite binding is a communication path between an arbitrary number of distributed component interfaces and is represented as a set of primitive bindings and binding components. Binding components are called *FRACTAL* connectors and are normal *FRACTAL* components, whose role is dedicated to communication [3]. As connectors are of no primary concern in *FRACTAL*, no further specification on how to interact is provided. This is contrary to the work proposed within this paper, where connectors play an important role in component interaction.

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 [6] under contract 809444.

References

1. D. Bálek. *Connectors in Software Architectures.* PhD thesis, Charles University Prague, Faculty of Mathematics and Physics; Department of Software Engineering, Feb. 2002.
2. D. Bálek and F. Plasil. Software connectors and their role in component deployment. In *DAIS*, pages 69–84, 2001.
3. E. Bruneton, T. Coupaye, and J.-B. Stefani. *The Fractal Component Model.* ObjectWeb. http://fractal.objectweb.org/specification/index.html.
4. T. Bures and F. Plasil. Communication style driven connector configurations. In *Lecture Notes in Computer Science*, volume 3026, pages 102–116, 2004.
5. Charles University Prague, Department of Software Engineering. *SOFA - Software Appliances.* http://nenya.ms.mff.cuni.cz/ projects/sofa/tools/doc/compmodel.html.
6. COMPASS. *Component Based Automotive System Software.* http://embsys.technikum-wien.at /projects/compass/index.html.
7. G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering.* Addison Wesley, 2001.
8. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
9. B. Meyer. The grand challenge of trusted components. In *ICSE*, pages 660–667, 2003.

10. Microsoft Corporation and Redmond, WA. *DCOM architcture*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp.
11. O. Nierstrasz and D. Tsichritzis, editors. *Object-Oriented Software Composition*. Object-Oriented Series. Prentice-Hall, Dec. 1995.
12. ObjectWeb. *FRACTAL*. http://fractal.objectweb.org/.
13. OMG. *UML 2.0 Superstructure Specification*, 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-04.
14. OMG. *CORBA Component Model Specification Version 4.0*, 2006. http://www.omg.org/docs/formal/06-04-01.pdf.
15. R. H. Reussner and H. W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In S. L. Ivica Crnkovic and J. Stafford, editors, *Workshop on Component-based Software Engineering Proceedings*, 2002.
16. S. Robert, A. Radermacher, V. Seignole, S. Gérard, V. Watine, and F. Terrier. The CORBA connector model. In *SEM*, pages 76–82, 2005.
17. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
18. Sun Microsystems. *Enterprise JavaBeans^{TM} Specification 2.1 Final Release 2*. http://java.sun.com/products/ejb/docs.html.
19. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Jan. 1998.