Architectural Design of Component Composition Using First Class Connectors

Asuman Sünbül[1]

Technische Universität Berlin
Computergestützte Informationssysteme (CIS)
Sekr.E-N7, Einsteinufer 17,
D-10587 Berlin, Germany
+49-30-31479463
asu@cs.tu-berlin.de*

Abstract. Component-oriented software development [7],[11] is becoming more and more important for the production of large-scale software applications. Today's systems are constructed by combining different software components rather than built from scratch. Provided *component interfaces* or informal descriptions are not enough for designing reliable software systems. Therefore we need precise description means for formulating component behavior without uncovering implementation details of the component. In this paper, we will present a model for the design of component-based software systems, where the design aspects of *connectors* between components play a central role. We will therefore introduce the concept of *connector refinement* allowing to abstract from technical connector implementations on the design level of a system.

1 Composition Techniques for Component Based Software Development

The degree of *reusability* not only depends on the facilities of the components themselves, but also on how the components are glued together and what kind of composition mechanisms have been used. The latter aspect has been neglected in the majority of projects dealing with component-based software engineering.

Current approaches tackle this problem on the technical layer, using e.g. CORBA, RPC, script languages or event channels etc.. These approaches have a strong emphasis on solving technical component interaction problems. The realization of the overall problem specification is covered by these low-level problems. Therefore, it is important to provide means that allow system design on a high level of abstraction. Our main goal is finding high level composition aspects in the design phase allowing an abstraction from the concrete technical realization [14]. We think, that a clear separation of concerns must be applied to the connectors in the same way as it is done for the development of the components themselves. In order to provide means for a high level composition method, we are going to discuss our understanding of composition. In [8],

* This research was supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316)

[13] component composition is the way of assembling software components to a composable software system by using interoperability and interconnection techniques. Interoperability deals with the cooperation aspects among components e.g. including the questions how components work together, which task have been done by which components etc.. Interconnection describes the coordinated communication among components, e.g. control flow among components, and which component is sending what kind of messages to which components.

In nearly all applications, the starting situation is that there are

- formal/informal descriptions of the target system;
- set of components serving as potential candidates for the composition but not fulfilling the exact definition, e.g many services, provided services cannot be used in a specific context;
- other technical mismatches.

There are different ways to solve these problems, using component based wrappers or component based adapters as suggested and constructed in [19] are one possiblity overcoming service mismatches or bringing services into the requiered form.

2 Connector Refinement

The idea behind connector refinement allows to actually *design* connectors in the same way as we design components: starting from abstract representations of connectors we stepwisely refine those finally ending up in concrete technical connector realizations.

In the design model we just draw arrows between components. Each of these arrows represents a design connector being subject to refinement. As shown in Figure 1, the design connector is refined to a more complex structure consisting of another three components being responsible for realizing the connector.

Using this concept, patterns of connector refinements can be defined covering commonly used connector types. These refinement pattern show how an abstract connector given on the design level can be refined to a specific connector realization. In the following, we will give examples, how connectors can be refined to more concrete connectors.

2.1 Event Channel Connectors

Components can post messages to the event channel. These messages have no direct recipient and can be taken from any components matching with its service specifications. In Figure 1, we have shown, that at the design level a component C_1 needs a service e from another component. At this time it is not clear, which component provides this service. While refining this connector an event-channel component receives the message coming from C_1 and passes this to a component providing the required service. This request presumes that the component must have registered itself at the event channel component with its services.

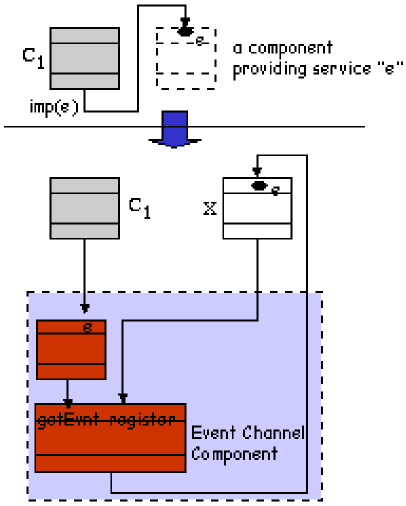

Fig. 1.

2.2 Tuple Spaces Connectors

Tuple spaces based composition is similar to blackboard composition. Components can ”stick” messages on it, which can be read by other components. After reading a message, a component can react or ignore this message. A tuple space provides basic operations for realizing the component communication (*readMsg, notifyComp, writeMsg, deleteMsg*, etc.)

Figure 2 illustrates the connector refinement between the component C_1 and C_2. The service call at the design level is refined into three more technical components. The component A has the task to pass the incoming messages from component A to the blackboard component. Component B checks the blackboard component whether there is a message for component C_2, in this case component B is delivering this message to C_2.

Tuple spaces based techniques are, for instance, realized in JiniTM or InfoBus technology [1, 12] and it is suitable for building robust and large-scale applications. Using this concept, patterns of connector refinements can be defined covering commonly used connector types. Of course, the scenarios described above are much more complex than explained here, in order to focus on the main idea, we have abstracted from some details.

2.3 A Formal Representation of Components and Connectors

This representation is necessary, because in many applications, the description of components is often reduced to the enumeration of the provided services together with the

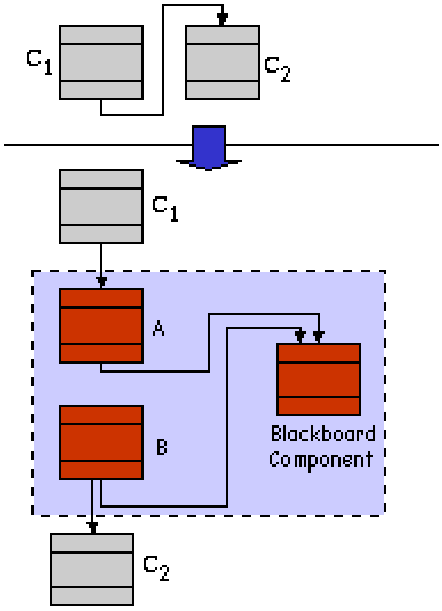

Fig. 2.

corresponding signatures. However, in most cases this is not sufficient to get all information about the *real behavior* of a component; important questions remain unanswered by this kind of description:

- Does the execution of a component service have any side effects that influences the behavior of other services on subsequent invocations? Answering this question is indispensable for building reliable software systems; without this information, the component is not completely specified.
- Does the component fit into a specific domain and are there any restrictions concerning the composition with other components? How can components being designed as general purpose components used in a specific domain?

Therefore we need precise description means for formulating component behavior without uncovering implementation details of the component [5]. In the following, a formal representation of the component model is presented in more detail. The model serves as a basis for formulating the structure of components and connectors, and the relationships between them. The formalization uses the notation of *Abstract State Machines* (ASMs) as introduced in [9]. ASM basically define states in terms of static algebras and transition rules being used to transform one state to another. We will use ASMs in the following way: The functions and universes describing a state will be used to describe the static structures of the components and connectors, while the ASM rules will be used to describe the consistency rules of our component model.

Representation of components. Components are modeled to be elements of a

universe $Component$

In ASMs, a universe stands for a basic set of elements. A component has a name represented as a string, which is modeled using a function *ComponentName* defined as follows:

function $ComponentName(c : Component) \rightarrow String$

For example, if a component c has the name “Library”, the following equation holds:

$$ComponentName(c) = \texttt{"Library"}$$

Export and import services of components are represented by universes as well:

universe $ExpService, ImpService$
universe $Service = ExpService \cup ImpService$

The fact that a component has a set of export and import services is modeled using the following functions:

functions $Exports(c : Component) \rightarrow \{ExpService\},$
$Imports(c : Component) \rightarrow \{ImpService\}$

For example, to express that a component c exports services ϵ_0, ϵ_1, and imports services ι_0, ι_1 the values of these functions would be as follows:

$$Exports(c) = \{\epsilon_0, \epsilon_1\}$$
$$Imports(c) = \{\iota_0, \iota_1\}$$

The services themselves have a name, a list of parameters each of which has a specific type and a mode describing whether it is an ”in”, ”out”, or ”inout” parameter, and an optional result type. Types are elements of a universe ”Type”; the concrete representation of types is not regarded in the model. However, it is assumed that a function ”*conforms_to*” exists that returns true, if the two type arguments are conform to each other.

universe $Type$
universe $ParameterMode = \{in, out, inout\}$
functions $ServiceName(s : Service) \rightarrow String,$
$ServiceParameter(s : Service)$
$\rightarrow [\ (t : Type, m : ParameterMode)\ ],$
$ServiceResType(s : Service) \rightarrow Type$
external function $conforms\_to(t_1 : Type, t_2 : Type) \rightarrow Boolean$

The function ”*conforms_to*” is an ”external” function which is not further specified on this abstraction level. External functions can be seen as ”oracles” for the formalization; it is assumed that they return the ”right” value for given parameters. Using this technique, one is able to focus on the main issues of a specification; functions marked as ”external” can be modeled in further refinement levels.

Abstraction from concrete Notations. Our component model abstracts from concrete notations for component usage constraints, service pre- and postconditions, and other specification items occurring in a concrete system specification. For each class of specification item, we therefore introduce a universe to express the existence of these kinds of items. The advantage of this abstraction is, that our model focuses on the questions arising when building systems using components and connectors without dealing with technical details of the specification techniques used in a specific system specification.

We therefore define the following universes signaling the existence of information needed to express the consistency rules of our component model:

universe $Constraint, Specification, UseStructure$

These universes are used for the following purposes:

Constraint A constraint formula expresses possible usages of the export services of a component. For example, if a component c exports services s_0, s_1, ..., s_n and $s_1, \ldots, s_n$ can only be used, if the initialization service s_0 has been invoked before, the component constraint formula would be the place to formulate this kind of condition. Examples for possible notations of component constraints are temporal logic formulas [20, 21], and path expressions [16].

Specification A specification expresses either the functionality of a service exported by a component or the required functionality of an imported service. No further assumption about the format and notation of this specification is made on the highest abstraction level. Examples for possible service specification formats are abstract data types [16], CSP [2], Z [15], or any other specification language that can be used to express functional behavior.

UseStructure A use structure expresses the usage of imported services in the body of the component. Typically, a use structure is given as a net describing possible orderings of service usages while executing exported services. A use structure must be matched against the constraints given by the components the services of which are used. In [20] and [21] import petri-nets are used to model the usage of imported services of the component.

The following function definitions bring the above universes and the previously introduced components and services of our model together:

functions $ComponentConstraint(c : Component) \rightarrow Constraint,$
$ProvidedSpec(s : ExpService) \rightarrow Specification,$
$RequiredSpec(s : ImpService) \rightarrow Specification,$
$ImportStructure(s : ExpService) \rightarrow UseStructure$

The function *containedServices* allows to extract the information which services are contained in a specific use structure.

external function $containedServices(u : UseStructure) \rightarrow \{ImpService\}$

This function is marked as "external", because we did not make any assumptions about the format of a use structure. In case the use structures are given as petri-nets, where the used services are modeled as transitions, the *containedServices* function could be an algorithm extracting the corresponding transitions from the petri-net.

Similarly, the following functions are required as externally given in order to express consistency conditions of our component model:

external functions
$impliesSpec(expspec : Specification, impspec : Specification) \rightarrow Boolean,$
$meetsConstraint(u : UseStructure, c : Constraint) \rightarrow Boolean$

The *impliesSpec* function is responsible for checking whether the specification of an import service S_I can be derived by the specification of an exported service S_E. In this case, a necessary condition is fulfilled that the exported service can be "plugged" in the component requiring service S_I. The *meetsConstraint* function is used to check whether the usage of imported services $s_1, \ldots s_n$ in the body of a component is consistent with the constraints specified for the components exporting $s_1, \ldots s_n$.

Representation of connectors. In general, connectors are responsible for interconnecting components. More precisely, a connector maps a required service S_A of a component A to an exported service S_B of a component B. In our model, this is represented as follows:

universe $Connector$
function $ServiceConnector(s_i : ImpService, s_e : ExpService) \rightarrow Connector$

Connectors are elements of the universe *Connector*. If a connector cn maps an import service S_A to an export service S_B, the following condition holds:

$$ServiceConnector(S_A, S_B) = cn$$

Component Model Consistency Rules. With the above introduced formalizations of component and connectors we have now the framework for expressing consistency rules for our component model. In this paper, we can only sketch this part of the formalization in order to give you an idea, how this kind of rules can be specified.

As an example, we specify the rules for checking whether the signatures of an import and an export service match. These rules are active, whenever a connection should be made between two components. We therefore first define a macro defining the formula for testing two service parameters on equality:

param_equal(P1,P2)
$(conforms\_to(P1\hat{}t, P2\hat{}t)) \ \wedge$
$(P1\hat{}m = P2\hat{}m)$

As one can see, the parameter are equal, if the types are conform to each other *and* the modes are equal. This macro is used in the following "asm" specifying the rules for signature matching:

asm $check\_signatures(s_{imp} : ImpService, s_{exp} : ExpService) \rightarrow Boolean$
  is
    function $\tau_1 \leftarrow ServiceParameter(s_{imp}),$
      $\tau_2 \leftarrow ServiceParameter(s_{exp})$
    if $(\tau_1 =\sim [\&p_1|\&pl_1]) \wedge (\tau_2 =\sim [\&p_2|\&pl_2])$ **then**
      if $\neg param\_equal(\&p_1, \&p_2)$ **then**
        return false
      else
        $\tau_1 := \&pl_1$
        $\tau_2 := \&pl_2$
      endif
    elseif $(\tau_1 = []) \wedge (\tau_2 = [])$ **then**
      if $\neg param\_equal(ServiceResType(s_{imp}), ServiceResType(s_{exp})$ **then**
        return false
      else
        return true
      endif
    else
      return false
    endif
endasm

Each of the parameters of the services are checked to be equal; whenever the parameter list differ, **false** is returned. The $check\_signatures$ function only returns **true**, if all parameters and the optional result parameters are equal.

2.4 Technical Realization and Integration into existing approaches

Finally, we will point out how this work can be integrated within existing component architectures JavaBeans[18], EnterpriseBeans[6] and DCOM [17]. We are currently working on a tool implementation [10] serving as a “composition assistant” for heterogeneous component architectures. The composition of different component technologies is predominantly implemented by proprietary solutions, generally enabling connection between certain architectures, e.g. ActiveX Bridges or JIntegra, facilitating compositions of COM and JavaBeans. There is practically no common generic solution for the composition of components between any architectures. Generic approaches, for instance ”interceptor approach”, aim to provide means for ”uncageing” components from necessary type transformations and call adjustments completely achieving a higher reusability of components. In contrast to techniques like IIOP, the interceptor approach facilitates more flexibility and reusability, because heterogeneous architectures can communicate without adjustments.

3 Conclusion

Following the current trends in the field of software engineering, component composition will be the central task for developers in near future. It is therefore important to provide efficient and powerful means to deal with the problems occurring when assembling software systems from existing components. In this paper, we have presented a basic model for components and connector that can be taken as basis for component-based development methods. Through the notion of *connector refinement* we have presented a possibility to apply well-establish design principles of software systems in general to component connectors. The latter aspect is often neglected in traditional software development methods; connectors are mostly just *implemented* , sometimes even as part of the integration test phase.

This paper does not describe the component model in full detail, because this would be out of the scope of this paper. Especially the formal representation of connector refinements is not contained in here; this aspect is currently under development and will be further elaborated in near future. Furthermore, we are working on an implementation of a (graphical) language for translating component structures to the formal models presented in Section 2.3. The consistency rules can be – due to the execution semantics of Abstract State Machines – directly be taken to implement a support system for our approach using ASM implementations like [3].

References

1. *JavaSpaces Principles, Patterns, and Practice*. Addison Wesley, 1999.
2. ALLEN, R., AND GARLAN, D. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering* (May 1994), pp. 71–80.
3. ANLAUFF, M. Xasm – an extensible, component-based abstract state machines language. In *Proceedings of the ASM 2000 Workshop* (2000). to appear.
4. ANLAUFF, M., KUTTER, P., PIERANTONIO, A., AND SÜNBÜL, A. Using domain-specific languages for the realization of component composition. In *Proceedings of the FASE 2000 Conference* (2000).
5. BATORY, D. Intelligent components and software generators. Tech. Rep. 97-06, Department of Computer Sciences, University of Texas at Austin, April 1997. Invited presentation to the Software Quality Institute Symposium on Software Reliability.
6. CODELLA, C. F., DILLENBERGER, D. N., FERGUSON, D. F., JACKSON, R. D., MIKALSEN, T. A., AND SILVA-LEPE, I. Support for Enterprise JavaBeans in Component Broker. *IBM Systems Journal 37*, 4 (???? 1998), 502–??
7. C.SZYPERSKI. *Component Software - Beyond Object Oriented Programming*. Addison Wesley, 1998.
8. GASH, B., KELTER, U., KOPFER, H., AND WEBER, H. Reference model for the integration of tools in the 'EUREKA software factory'. In *ACM-IEEE Fall Joint Comp. Conf* (Oct. 1987).
9. GUREVICH, Y. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1995, pp. 9–36.
10. LEICHER, A. Konzepte zur Realisierung von Interoperabilität zwischen heterogenen Komponentenarchitekturen. Diplomandencolloquim, Continious Software Engineering AG, ISST Berlin, July 2000.
11. MENS, K., LOPES, C., TEKINERDOGAN, B., AND KICZALES, G. Aspect-oriented programming. *Lecture Notes in Computer Science 1357* (1998), 483–??
12. MICROSYSTEMS, S. Jini [tm] - connection technology. http://java.sun.com/products/.
13. MONROE, R. T., GARLAN, D., AND WILE, D. ACME Straw Manual. Tech. rep., Carnegie Mellon University, Pittsburgh, Nov. 1997. Version 0.1.1.

14. MÜLLER, H., AND WEBER, H., Eds. *Continuous Engineering of Industrial-Scale Software Systems* (Mar. 1998). Proc. Dagstuhl Seminar, to appear.
15. ROYCE, W., AND ROYCE, W. Software architecture: Integrating process and technology. *TRW Space and Defense* (1991).
16. SCHUMANN, H., AND GOEDICKE, M. Component-oriented software development with Π. ISST-Berichte 21/94, ISST, Sept. 1994.
17. SESSIONS, R. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Son, 1997.
18. SUN MICROSYSTEMS. *Java Beans(TM)*, July 1997. Graham Hamilton (ed.). Version 1.0.1.
19. SÜNBÜL, A. *Architectural Design of Evolutionary Software Systems: A Concept for the Compostion of Components*. PhD thesis, Technical University Berlin, April 1999.
20. WEBER, H., PADBERG, J., AND SÜNBÜL, A. Petri net based components for evolvable architectures. IEEE International Conference on Systems Integration.
21. WEBER, H., SÜNBÜL, A., AND PADBERG, J. Modelling information and communication infrastructures for evolutionary design of busines processes using component technologies. Society for Design and Process Science, IEEE International Conference on Systems Integration.