

Towards Component Based Systems: Refining Connectors

Matthias Anlauff^{1,2} Asuman Sünbül^{1,3}

*Kestrel Institute
Palo Alto, California, USA*

Abstract

Component-oriented software development 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 Introduction: Component Based Systems in Software Architecture

During the last few years, component-based software development (CBSD) has evolved to one of the most popular sub-disciplines of software engineering. But, 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 has 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

¹ This material is based upon work supported by the Air Force Research Laboratory under Contract No. F30602-00-C-0209 and F30602-01-C-0123.

² Email: ma@kestrel.edu

³ Email: asu@kestrel.edu

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. 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 this work, a software development approach focusing on component composition techniques via connectors is proposed. The semantics of the component model forming the technical basis for the approach is specified using the Abstract State Machines (ASM) [5] method. Additionally, a component model input language (CMIL) is introduced as a user interface to the component model. In CMIL, system design processes are carried out using refinement techniques, especially by connector refinements. Therefore, the notion of abstract connectors is introduced as a means to abstract from concrete connector specifications. The semantics of the refinement techniques is also fully formalized using ASMs. As a side effect, the formalizations contained in the work have directly been used to generate an executable version of the ASM rules using an ASM support environment. The generated code, the "CMIL-checker", has also been used to validate the examples and case studies presented in different projects [9].

1.1 A Formal Representation of Components and Connectors

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 [?]. 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:

$$\mathit{ComponentName}(c) = \text{"Library"}$$

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

$$\begin{aligned} &\mathbf{universe} \quad \mathit{ExpService}, \mathit{ImpService} \\ &\mathbf{universe} \quad \mathit{Service} = \mathit{ExpService} \cup \mathit{ImpService} \end{aligned}$$

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

$$\begin{aligned} &\mathbf{functions} \quad \mathit{Exports}(c : \mathit{Component}) \rightarrow \{\mathit{ExpService}\}, \\ &\quad \quad \quad \mathit{Imports}(c : \mathit{Component}) \rightarrow \{\mathit{ImpService}\} \end{aligned}$$

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:

$$\begin{aligned} \mathit{Exports}(c) &= \{\epsilon_0, \epsilon_1\} \\ \mathit{Imports}(c) &= \{\iota_0, \iota_1\} \end{aligned}$$

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.

$$\begin{aligned} &\mathbf{universe} \quad \mathit{Type} \\ &\mathbf{universe} \quad \mathit{ParameterMode} = \{in, out, inout\} \\ &\mathbf{functions} \quad \mathit{ServiceName}(s : \mathit{Service}) \rightarrow \mathit{String}, \\ &\quad \quad \quad \mathit{ServiceParameter}(s : \mathit{Service}) \\ &\quad \quad \quad \rightarrow [(t : \mathit{Type}, m : \mathit{ParameterMode})], \\ &\quad \quad \quad \mathit{ServiceResType}(s : \mathit{Service}) \rightarrow \mathit{Type} \\ &\mathbf{external\ function} \quad \mathit{conforms_to}(t_1 : \mathit{Type}, t_2 : \mathit{Type}) \rightarrow \mathit{Boolean} \end{aligned}$$

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, \dots, s_n and s_1, \dots, 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 [11,12], and path expressions [8].

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 [8], CSP [1], Z [7], 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 [11] and [12] 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 *con-*

tainedServices 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*) → *Boolean*,
meetsConstraint(*u* : *UseStructure*, *c* : *Constraint*) → *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, \dots, s_n in the body of a component is consistent with the constraints specified for the components exporting s_1, \dots, 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*) → *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:

$$\ulcorner \text{param_equal}(P1, P2) \frac{}{(conforms_to(P1 \hat{t}, P2 \hat{t})) \wedge (P1 \hat{m} = P2 \hat{m})}$$

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(simp : ImpService, sexp : ExpService) → Boolean
is
  function  $\tau_1 \leftarrow \text{ServiceParameter}(s_{imp}),$ 
            $\tau_2 \leftarrow \text{ServiceParameter}(s_{exp})$ 
  if ( $\tau_1 =_{\sim} [\&p_1|\&pl_1]$ )  $\wedge$  ( $\tau_2 =_{\sim} [\&p_2|\&pl_2]$ ) then
    if  $\neg\text{param\_equal}(\&p_1, \&p_2)$  then
      return false
    else
       $\tau_1 := \&pl_1$ 
       $\tau_2 := \&pl_2$ 
    endif
  elseif ( $\tau_1 = []$ )  $\wedge$  ( $\tau_2 = []$ ) then
    if  $\neg\text{param\_equal}(\text{ServiceResType}(s_{imp}), \text{ServiceResType}(s_{exp}))$  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 Representing Standard Connector Techniques using CMIL

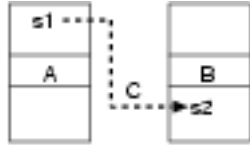
We're introducing connector refinement as a powerful means to express relationships between an abstract system architecture and more refined versions of it.

By the introduction of abstract connectors, it is possible, to abstract from actual connector implementations in the same way as it is possible to abstract from component realizations by only specifying their export interfaces. In the sequel, we will show how standard connector techniques – like *Wrapper* and *Adapter* etc. – are expressed using in CMIL using connector refinement.

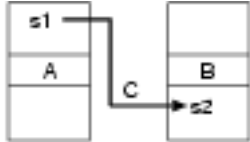
In many cases, we will only use the service names without specifying their signatures and assume, that the introduced connectors are consistent with respect to the consistency rules of the component model. We will also make use of graphical representations for describing CMIL components and connectors, where the symbols have the following meaning:



stands for a component with export and import interface. The interfaces contain the signatures of the exported/imported services.



stands for an “abstract” connector C connecting the export service s_1 from component A with the import service s_2 from component B .



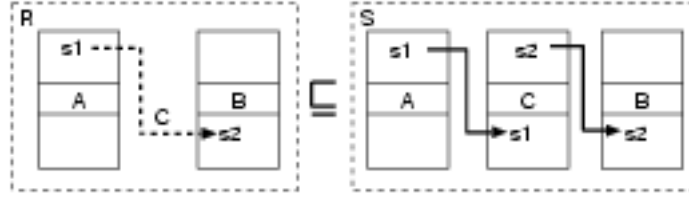
same as the above, only that the connector is a concrete one. In this case, the connector name may be omitted.

Systems as defined in CMIL are represented as dashed boxes with the name of the system in the top left corner.

3 Adapters

A common technique of defining connections between components is the use of “adapter” code for adjusting the provided exported services of a component A to fit to the required import services of a component B . The functionality of the adapter is usually not very complex and is mainly used for performing data type conversions, changing the order of parameters, filtering input and output parameters etc. . . One the one hand, this adapter code is very important to technically compose system components, but on the other hand it should be possible to abstract from this kind of adapter components when presenting the architectural description of a system.

In CMIL, abstract connectors can be used to achieve this effect, by introducing in an abstract system R a abstract connector C representing a connector between two non-matching services s_1 and s_2 . In system S the abstract connector C is then refined by a component providing the required export service s_2 and importing the exported service s_1 . The corresponding CMIL description is given as follows:


 Fig. 1. Connector Refinement using an adapter component C

<pre> SYSTEM R { COMPONENT A { EXPORTS { s_1 } } COMPONENT B { EXPORTS { ... } IMPORTS { s_2 } } CONNECTOR C { $A :: s_1$ → [] → $B :: s_2$ } } </pre>	\sqsubseteq	<pre> SYSTEM S REFINES R { COMPONENT C { EXPORTS { s_2 } IMPORTS { s_1 } } CONNECTOR { $A :: s_1$ → [C] → $B :: s_2$ } } </pre>
--	---------------	---

A corresponding graphical representation of this connector refinement example is contained in Figure 1.

4 Wrapper Connectors

Wrappers are a very common technique in solving interoperation problems. This approach avoids direct modification of the components and is more flexible. A wrapper encapsulates a component, in order to make it usable in some new way for another component. Wrappers can be used to expose all or some of the functionality of the component they are wrapping. In CMIL, this situation can be modeled by introducing a wrapper component as follows:

$$\begin{array}{l}
 \text{SYSTEM } \mathbf{R} \{ \\
 \quad \text{COMPONENT } \mathbf{A} \{ \\
 \quad \quad \text{EXPORTS } \{ s_1 \} \\
 \quad \quad \text{IMPORTS } \{ s_2 \} \\
 \quad \} \\
 \quad \text{COMPONENT } \mathbf{B} \{ \\
 \quad \quad \text{EXPORTS } \{ \dots \} \\
 \quad \quad \text{IMPORTS } \{ s_3 \} \\
 \quad \} \\
 \quad \text{COMPONENT } \mathbf{C} \{ \\
 \quad \quad \text{EXPORTS } \{ s_4 \} \\
 \quad \} \\
 \quad \text{CONNECTOR } \mathbf{W} \{ \\
 \quad \quad A :: s_1 \rightarrow [] \rightarrow B :: s_3 \\
 \quad \} \\
 \quad \text{CONNECTOR } \mathbf{W} \{ \\
 \quad \quad C :: s_4 \rightarrow [] \rightarrow A :: s_2 \\
 \quad \} \\
 \} \\
 \end{array}
 \quad \sqsubseteq \quad
 \begin{array}{l}
 \text{SYSTEM } \mathbf{S} \text{ REFINES } \mathbf{R} \{ \\
 \quad \text{WRAPPER-COMPONENT } \mathbf{W} \{ \\
 \quad \quad \text{EXPORTS } \{ \\
 \quad \quad \quad s_3 \\
 \quad \quad \quad s_2 \\
 \quad \quad \} \\
 \quad \quad \text{IMPORTS } \{ \\
 \quad \quad \quad s_1 \\
 \quad \quad \quad s_4 \\
 \quad \quad \} \\
 \quad \text{CONNECTOR } \{ \\
 \quad \quad A :: s_1 \rightarrow [W] \rightarrow B :: s_3 \\
 \quad \} \\
 \quad \text{CONNECTOR } \{ \\
 \quad \quad C :: s_4 \rightarrow [W] \rightarrow A :: s_2 \\
 \quad \} \\
 \} \\
 \end{array}$$

In this case, the component A exports a service, which is used by B and requires a service, which is exported by C . In system R , we have modeled this using abstract connectors, because we assume, the the services do not fit together without wrapping A . In the refined system S , such a wrapper component W has been introduced. It refines the two abstract connectors from system R by providing suitable export and import services. The corresponding graphical representation of this connector refinement example is given in Figure 2. ⁴

5 Blackboard Connectors

Blackboards basically provide the possibility to attach and to query for messages for a certain receiver. In CMIL, a simplified blackboard component would look like this:

$$\begin{array}{l}
 \text{COMPONENT } \mathbf{Blackboard} \{ \\
 \quad \text{EXPORTS } \{ \\
 \quad \quad \text{void } attach(Receiver, Message) \\
 \quad \quad [Message] query(Receiver) \\
 \quad \} \\
 \} \\
 \end{array}$$

⁴ In this picture, component A is drawn inside the wrapper component W just for illustration purposes; A could have been drawn outside the W component without changing the semantics of the picture.

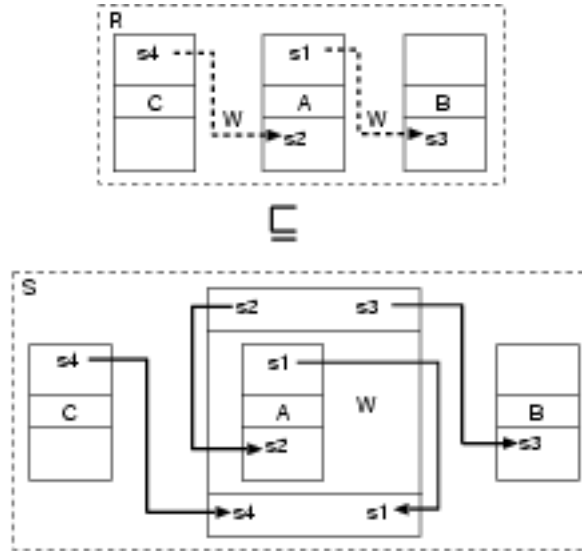


Fig. 2. Connector refinement using a wrapper component W

assuming that the types *Receiver* and *Message* are defined. Assuming, we have given the abstract system description R as shown below, then we could refine the C connector using the blackboard and a suitable wrapper component as follows:

<pre> SYSTEM R { COMPONENT A { EXPORTS {void $s_1()$ } } COMPONENT B { EXPORTS { ... } IMPORTS {void $s_2()$ } } CONNECTOR C { $A :: s_1 \rightarrow [] \rightarrow B :: s_2$ } } </pre>	\sqsubseteq	<pre> SYSTEM S REFINES R { TYPES {<i>Receiver Message</i>} INCLUDE <i>Blackboard</i> COMPONENT Attach { EXPORTS { s_2 } IMPORTS { $attach(Receiver, Message)$ } } COMPONENT Poll { IMPORTS { s_1 $[Message] query(Receiver)$ } } COMPONENT C { <i>Attach Poll</i> } CONNECTOR {<i>Blackboard</i> :: $attach$ $\rightarrow C :: attach$} CONNECTOR {<i>Blackboard</i> :: $query$ $\rightarrow C :: query$} CONNECTOR { $A :: s_1 \rightarrow [C] \rightarrow B :: s_2$ } } </pre>
---	---------------	--

The wrapper component C consists of two sub-components: *Attach* and *Poll*. Whenever B send a message through its import service s_2 the export service s_2 of *Attach* is invoked. It attaches a messages on the blackboard using the corresponding export service of the component *Blackboard*. The *Poll* component periodically polls for messages for A using the *query* service, and as soon as there is a message attached at the blackboard it invokes the corresponding export service of A . Figure 3 contains the graphical representation of this connector refinement example.

6 Request Broker Connectors

The main purpose of “Request Brokers” is to provide connection services between different software entities, where the location of these entities is not restricted to any pre-determined configuration. That means that the entities using the request broker services may run on the same computer, or are distributed over a network and link together using technical communication protocols. A popular example of such an request broker component is the *Object Request Broker* of the CORBA standard [6].

In CMIL, this kind of connectors can on the one hand be directly specified, and on the other hand possible refinements of this kind of connectors can be

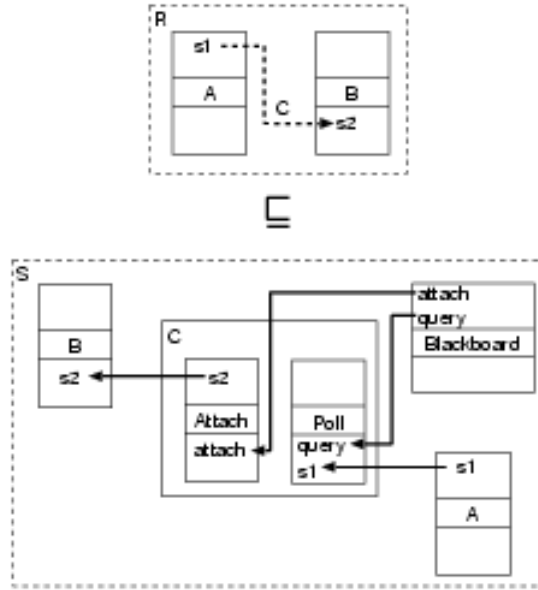


Fig. 3. Connector refinement using a blackboard component

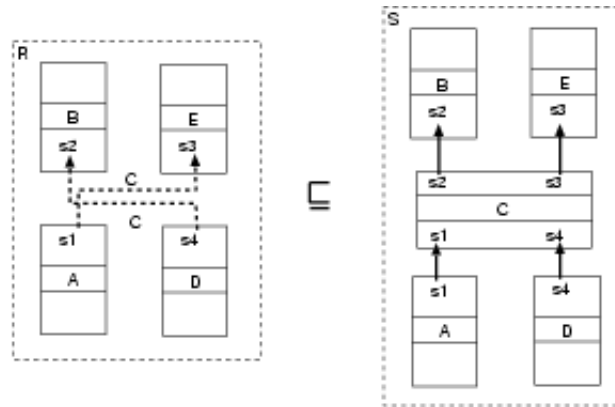


Fig. 4. Representation of an abstract request broker connector and its refinement given using the expressive means of the language. As an example, we consider the four component A , B , D , and E in Figure 4. In the abstract system R on the left-hand-side of this figure they are connected with an abstract connector C consisting of two different connections between the import services of B and E and the export services A and D . In this way, we can regard the abstract connector C as a request broker connector, in the same sense as described above. In the refined system S the realization of the connector is given by specifying a request broker component C providing the services $s2$ and $s3$ and requiring the services $s1$ and $s4$ according to the interfaces specified in the abstract system.

In Figure 5 the corresponding textual version of the refinement is given using the syntax of the CMIL.

<pre> SYSTEM R { COMPONENT A { EXPORTS { s_1 } } COMPONENT B { IMPORTS { s_2 } } COMPONENT E { IMPORTS { s_3 } } COMPONENT D { EXPORTS { s_4 } } CONNECTOR C { $A :: s_1 \rightarrow [] \rightarrow E :: s_3$ } CONNECTOR C { $D :: s_4 \rightarrow [] \rightarrow B :: s_2$ } } </pre>	\sqsubseteq	<pre> SYSTEM S REFINES R { REQUESTBROKER-COMPONENT C { EXPORTS { s_2 s_3 } IMPORTS { s_1 s_4 } } CONNECTOR { $A :: s_1 \rightarrow [C] \rightarrow E :: s_3$ } CONNECTOR { $D :: s_4 \rightarrow [C] \rightarrow B :: s_2$ } } </pre>
--	---------------	---

Fig. 5. Refinement of an abstract Request Broker Connector in CMIL

7 Conclusion

In this paper, we have presented a set of examples of connector refinement in order to illustrate how the refinement techniques of the component model can be used to approach the concrete realization of a component-connector-system step by step: First you define the general interconnections between the components using abstract connectors, then you refine these connectors to standard connector techniques, as shown in this paper, and finally these refined connectors can be used to directly derive executable program code.

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-established 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. A well-elaborated version can be found in [9].

Furthermore, we are working on an implementation of a (graphical) language for translating component structures to the formal models presented in Section 1.1. 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 [2].

References

- [1] ALLEN, R., AND GARLAN, D. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering* (May 1994), pp. 71–80.
- [2] ANLAUFF, M. Xasm – an extensible, component-based abstract state machines language. In *Proceedings of the ASM 2000 Workshop* (2000), vol. 1912 of *Lecture Notes in Computer Science*.
- [3] 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).
- [4] GARLAN, D., AND SHAW, M. *Software Architecture: Perspectives On An Emerging Discipline*. Prentice Hall, 1995.

- [5] HUGGINS, J. The ASM homepage, "http://www.eecs.umich.edu/gasm/community.htm".
- [6] OBJECT MANAGEMENT GROUP INC. Corba components, joint revised submission. Tech. rep., OMG TC Document, July 2000.
- [7] ROYCE, W., AND ROYCE, W. Software architecture: Integrating process and technology. *TRW Space and Defense* (1991).
- [8] SCHUMANN, H., AND GOEDICKE, M. Component-oriented software development with II. ISST-Berichte 21/94, ISST, Sept. 1994.
- [9] SÜNBL, A. *Architectural Design of Evolutionary Software Systems in Continuous Software Engineering*. PhD thesis, Technical University Berlin, April 2001. DAV, ISBN 3-935316-84-4.
- [10] SZYPERSKI, C. *Component Software - Beyond Object Oriented Programming*. Addison Wesley, 1998.
- [11] WEBER, H., PADBERG, J., AND SÜNBL, A. Petri net based components for evolvable architectures. IEEE International Conference on Systems Integration.
- [12] WEBER, H., SÜNBL, A., AND PADBERG, J. Modelling information and communication infrastructures for evolutionary design of business processes using component technologies. Society for Design and Process Science, IEEE International Conference on Systems Integration.