Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 160 (2006) 75–96
www.elsevier.com/locate/entcs

A Component Model for Architectural Programming

Hubert Baumeister[1,6] Florian Hacklinger[2,6] Rolf Hennicker[3,6]
Alexander Knapp[4,6] Martin Wirsing[5,6]

Institut für Informatik
Ludwig-Maximilians-Universität München
Germany

Abstract

Software architectures and modular composition help in constructing large-scale software systems. Current programming languages provide only insufficient support for software architecture. "Architectural programming" overcomes the problem of architectural erosion in implementations by integrating concepts of software architecture into programming languages. We present the new programming language JAVA/A as an instance for Java-based architectural programming and show how JAVA/A integrates architectural notions such as components, connectors, and assemblies into Java. A main asset of JAVA/A is its underlying abstract component model which provides the basis for reasoning about software components and assemblies. We give a formalisation of the abstract component model in terms of transition systems and states as algebras, and prove a consistency result for assemblies.

Keywords: Architectural programming, software architecture, semantics of components

1 Introduction

According to the ANSI/IEEE 1471 2000-standard, software architecture is: "The fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution." [9] (cf. also, e.g., [5,17]). Therefore, components are modelled "through-

[1] Email: baumeister@ifi.lmu.de
[2] Email: hacklinger@ifi.lmu.de
[3] Email: hennicker@ifi.lmu.de
[4] Email: knapp@ifi.lmu.de
[5] Email: wirsing@ifi.lmu.de
[6] This research has been partially supported by the EC 6th Framework project SENSORIA "Software Engineering for Service-Oriented Overlay Computers" (IST 016004) and the GLOWA-Danube project (01LW0303A) sponsored by the German Federal Ministry of Education and Research.

doi:10.1016/j.entcs.2006.05.015

out the development life cycle and successively refined into deployment and runtime" [12].

Architecture description languages (ADLs), like Wright, Darwin, or Rapide, provide means to model and analyse both the static and dynamic properties of software architectures. ADL models, however, are mainly employed in the analysis and design steps of the classical software development phases. The implementation step is, at best, only supported by code generation facilities (for an ADL comparison and overview, see, e.g., [11]). Common implementation environments are not capable of representing architectural notions properly. Thus, a hand-coded implementation from an ADL model inevitably tends to loose its connection to the intended architectural structure during the maintenance steps; the same issue will hold for generated code tampered with by hand when not strictly adhering to the model-driven development discipline. The result is "architectural erosion" [14].

In order to counter architectural erosion in the implementation and maintenance phases, we propose the inclusion of architectural notions, like components, ports with provided and required interfaces as well as protocols, connectors, and assemblies, into a programming language and we call such a language an *architectural programming language*.

We present the new programming language JAVA/A [7] as an instance for a Java-based architectural programming language, show how JAVA/A integrates architectural notions into Java, and present the abstract component model which forms the semantical basis underlying JAVA/A. In JAVA/A components are strongly encapsulated behaviours of which only the exchange of messages with their environment according to well-defined provided and required operation interfaces can be observed. Hence, component reuse and replaceability is fostered. The component interfaces are bound to ports which regulate message exchange by protocols and ports can be linked by connectors establishing a communication channel between their owning components. Thus, safe communication can be specified and verified. A set of components whose ports are linked by connectors forms an assembly. The number of ports a component offers, the linkage of ports between components, and the number of components in an assembly can vary dynamically, providing basic means for dynamic reconfiguration. JAVA/A is supported by a compiler which translates JAVA/A programs to Java classes and includes the possibility to check port protocols for compatibilty, i.e., that two connected ports will only exchange messages the communication partner understands, and the trace of messages will not lead to a deadlock.

It may be noted that traditional component frameworks like EJB, JavaBeans, COM+, CORBA, &c. do not qualify as architectural programming languages, since these techniques do not properly support the concepts of (required) interfaces, connectors, and assemblies. However, the inclusion of interface protocols into programming languages has been discussed for quite a long time [20,13]. On the other hand, ArchJava [1], an architectural extension of Java, introduces the structural features of architectural programming languages, such as components and ports, into a programming language, but does not give any support for controlling the

dynamic behaviour of ports by e.g. port protocols.

A main asset of JAVA/A is its underlying abstract component model which has strongly influenced the design of JAVA/A and which provides the basis for reasoning on assemblies and port compatibility. The abstract component model is formalised using the interface automata approach [2] to I/O-transition systems for describing the observable behaviour and the "states-as-algebras" approach [4] for representing the internals of components and assemblies. In particular, component and port types are modelled as sorts in order to ease dynamic reconfiguration and dynamic creation and deletion.

The paper is structured as follows: In Sect. 2, we introduce architectural programming with JAVA/A, present the component metamodel of JAVA/A, illustrate the architectural concepts by an example and show how dynamic reconfiguration at runtime is supported by JAVA/A. In Sect. 3 we present the semantic basis for the main constituents of our component model and give mathematical formalisations of components, ports, and assemblies. As an example of a consistency result, we prove in Sect. 4 that model checking deadlock-freedom of port protocols induces deadlock-freedom of the corresponding assembly, provided that its components correctly refine the respective port protocol state machines. In the remaining sections 5 and 6 we compare our approach with related work and discuss ongoing and future work.

2 Architectural Programming

The basic idea of architectural programming is to preserve a software architecture throughout the software development process in a decent way. For instance, in the analysis phase the components of an architecture are implied by the system boundaries of a use case model. In the design, architectures may be represented with UML component diagrams or by a model in an architecture description language. In the implementation, an architecture should be implemented in an architectural programming language, like JAVA/A.

After giving a brief introduction to JAVA/A and its features, we illustrate the language by means of a (simplified) Bank–ATM example. Moreover, we discuss the reconfiguration possibilities offered by JAVA/A.

2.1 JAVA/A

Figure 1 summarises the component metamodel of JAVA/A. A component has ports, and each port has a provided and required interface and may have a protocol associated with it to describe the sequence of messages allowed for that port. A component is either a simple component or a composite component. A composite component consists of an assembly which contains components and connectors. A connector links two components by connecting ports they own. In order to support strong encapsulation of components, a composite component is not directly composed out of subcomponents and connectors, instead a composite component is built from an assembly by hiding the internal structure of how the composite

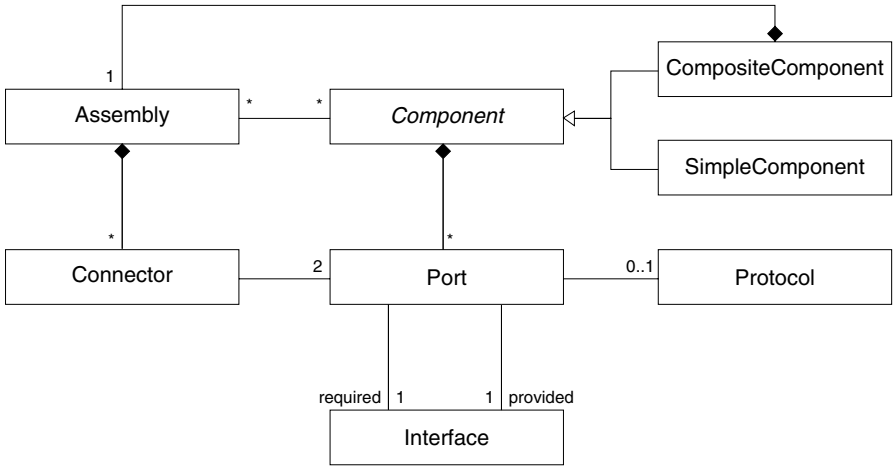


Figure 1. The component metamodel of the architectural programming approach.

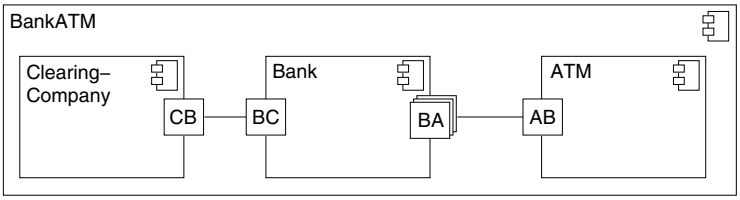
component is constructed.

JAVA/A extends Java by providing support for these architectural concepts, introducing keywords `port`, `required`, `provided`, `simple` and `composite component`, and `assembly`, and including port protocol descriptions as UML state machines. The JAVA/A compiler transforms JAVA/A components into pure Java code which can be compiled to byte code using the Java compiler. The generated Java classes are integrated into the JAVA/A component framework, which provides general operations that are common to all JAVA/A components, serving, for instance, for reconfiguration support. Each component can be compiled and deployed on its own, since the component's dependencies on the environment are encapsulated in ports. The correctness of an assembly, i.e., that for every connected port the required interface is satisfied by the provided interface of the connected counterpart and their protocols are compatible, can be assured using the UML state machine model checker HUGO [10], which is integrated with the JAVA/A compiler.

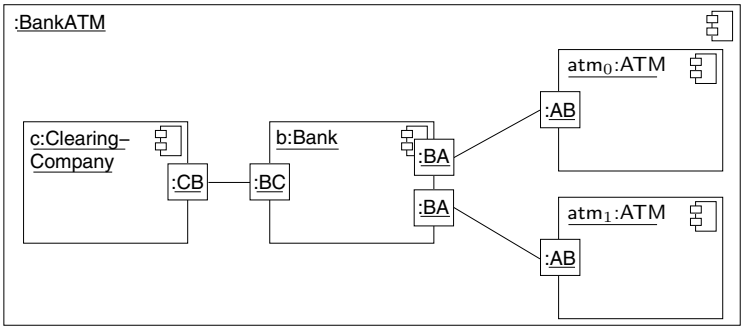
2.2 Example: Bank–ATM

We will illustrate our approach to architectural programming and the programming language JAVA/A by a simplified Bank–ATM example. In this example, a varying number of ATMs may be connected to a bank. An ATM can send an IBAN and a PIN to the bank in order to withdraw money. Then the bank asks a clearing company whether the IBAN together with the PIN is valid. If this is the case the ATM can withdraw an amount of money.

The design of the Bank–ATM system is shown in the component diagram in Fig. 2(a) where the composite component BankATM contains an assembly of three components ClearingCompany, Bank, and ATM whose ports are wired by appropriate connectors. The stacked boxes depicting port BA of component Bank indicates that it is a dynamic port which can have an arbitrary number of port instances. In contrast, static ports (like CB, BC and AB) must have a single instance at any time. Fig. 2(b) shows an admissible system configuration with two ATM instances whose



(a) UML 2.0 component diagram.



(b) An admissible system configuration.

Figure 2. Design model of the Bank–ATM example.

ports (of type AB) are linked to different port instances of type BA.

Port protocols are specified with UML state machines. A protocol describes the order and dependencies of messages which are sent and received by a port. Figure 3 displays the protocol of the port BC which specifies that each time the bank sends a message verifyPIN via BC it will wait for either a response pinOK or pinNotOk at the port BC before it can send another message verifyPIN.

2.3 Implementation of the Bank–ATM example using JAVA/A

The following code shows parts of the JAVA/A-declaration of the component Bank:

```
simple component Bank {
  Map balance = new HashMap();
  Queue pending = new LinkedList();
  BA current = null;
  Set verifieds = new HashSet();

  dynamic port BA {
    provided {
      signal verifyPIN(IBAN iban, int pin);
      signal withdraw(IBAN iban, Money amount);
    }
    required {
      void pinOk();
```

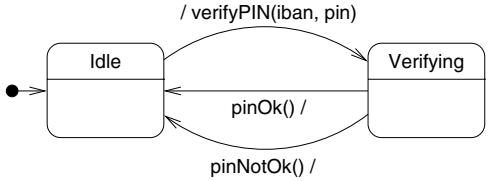


Figure 3. UML state machine describing the protocol of port BC.

```
      void pinNotOk();
      void withdrawOk();
      void withdrawNotOk();
    }
    <! // protocol of BA ... !>
  }

  port BC {
    provided {
      void pinOk();
      void pinNotOk();
    }
    required {
      void verifyPIN(IBAN iban, int pin);
    }

    <! // protocol of BC
      states {
        initial Initial;
        simple Idle;
        simple Verifying;
      }

      transitions {
        Initial -> Idle;
        Idle -> Verifying { effect verifyPIN(); }
        Verifying -> Idle { trigger pinOk(); }
        Verifying -> Idle { trigger pinNotOk(); }
      }
    !>
  }
  ...
}
```

In lines 2–5 component attributes are declared and initialised. The attributes of the component hold account balances (`balance`), a queue of verifyPIN requests (`pending`), the port instance of the currently processed request (`current`), and a set of port instances with already verified IBANs and PINs (`verifieds`). In lines 6–44, the ports `BA` and `BC` are defined. Each port declaration contains a set of provided operations (forming the provided interface) and a set of required operations (forming the required interface). Provided operations with the keyword `signal` are asynchronous, all other operations are synchronous. Port protocols are specified by UML state machines which are textually represented using the notation UTE [8]. For instance, lines 30–43 show the UTE represention of the UML state machine of port `BC` (see Fig. 3).

Every operation which is declared in a provided interface of a port must be implemented in the body of the port's owning component. For instance, the operation `verifyPIN` provided by the port `BA` is implemented as follows:

```
void verifyPIN(BA incoming, IBAN iban, int pin)
         implements BA.verifyPIN(IBAN, int) {
  pending.offer(new Object[]{incoming, iban, pin});
  synchronized (pending) {
    while (current != null) {
      try { pending.wait(); }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    Object[] request = (Object[])pending.poll();
    current = (BA)request[0];
    IBAN iban = (IBAN)request[1];
    int pin = ((Integer)request[2]).intValue();
    try {
      BC.verifyPIN(iban, pin);
    }
    catch (javaa.exception.ConnectionException e) {
      e.printStackTrace();
```

```
20      }
    }
  }
```

The parameter list of the implementing method is extended by a parameter `incoming` of the port type `BA` in order to allow the implementer to distinguish the source of a message in case of a dynamic port with multiple instances. To validate a PIN to an IBAN, the bank uses the clearing company which is connected to the bank at the port `BC`. Requests to verify a PIN on port `BC` (l. 16) must be sequentialised to comply with the protocol of `BC` (Fig. 3), hence the queue `pending` is used to keep requests that cannot be processed immediately. In case the port is not connected, the invocation of an operation declared in the port's required interface results in throwing a `ConnectionException`.

The implementation of the operation `pinOk` provided by the port `BC` reads as follows.

```
void pinOk() implements BC.pinOk() {
  verifieds.add(current);
  current.pinOk();
  current = null;
5 synchronized (pending) {
    pending.notify();
  }
}
```

As the component `Bank` will always have exactly one port instance of the type `BC` it is not necessary to extend the parameter list of `pinOk` by a parameter of type `BC`. First, the port instance of type `BA` of the currently processed request which is held in the field `current` is added to the set of already verified port instances. Next, a message `pinOk` is sent out via the port instance `current`. Finally, waiting threads are notified of the termination of the current verification request.

The JAVA/A realisation of the component `BankATM` is shown in the following code fragment:

```
composite component BankATM {
  assembly {
    component types { ATM, Bank, ClearingCompany }
    connector types {
5     Bank.BA, ATM.AB;
      ClearingCompany.CB, Bank.BC;
    }

    initial configuration {
10    ATM atm0 = new ATM();
      ATM atm1 = new ATM();
      Bank bank = new Bank();
      ClearingCompany cc = new ClearingCompany();
      Connector cn0 = new Connector();
15    cn0.connect(atm0.AB, bank.BA);
      Connector cn1 = new Connector();
      cn1.connect(atm1.AB, bank.BA);
      Connector cn2 = new Connector();
      cn2.connect(bank.BC, cc.CC);
20  }
  }
}
```

The component `BankATM` contains an assembly and therefore in l. 3–7 all component and connector types which will be used in the initial configuration and in future reconfigurations are declared (according to the component diagram given in Fig. 2(a)). The initial configuration (see Fig. 2(b)) is built up in lines 9–20. The connector type `Connector` is provided by the JAVA/A framework and realises

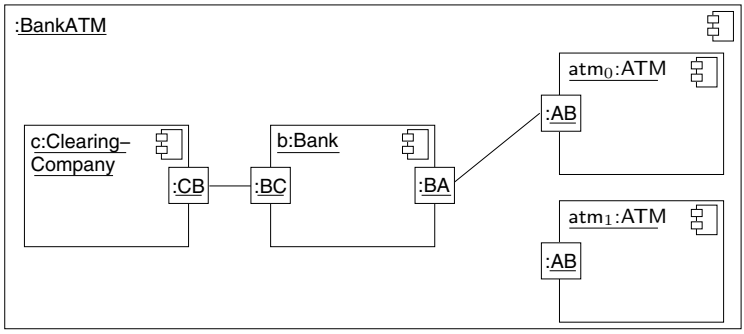


Figure 4. A configuration with a disconnected ATM.

message transportation with local procedure calls. The framework provides also a SOAP-based RPC connector.

2.4 Reconfiguration with JAVA/A

The term *dynamic reconfiguration* summarises changes to a component-based system at runtime, concerning creation and destruction of components and building up and removing connections between ports. JAVA/A supports each of these reconfiguration variants.

A possible reconfiguration in the Bank–ATM example is the connection and disconnection of ATMs. An idle ATM disconnects from the Bank and reconnects whenever a customer enters his bank card. Figure 4 shows a configuration where one ATM ($\mathtt{atm}_1$) is disconnected from the bank. When a customer enters his bank card the ATM executes the following code which realises the (re)connection of an ATM to the bank.

```
try {
  Component bank = componentLookUp(this, "Bank");
  Port ba = bank.getPort("BA");
  ConnectionRequest cr =
5     new ConnectionRequest(this, this, AB,
                            bank, ba, new Connector());
  reconfigurationRequest(cr);
}
catch (ReconfigurationException e) { ... }
```

The JAVA/A framework provides the operations `componentLookUp`, `getPort`, and `reconfigurationRequest`. The first retrieves a component using its identifier, `getPort` fetches a port instance from a component, and the last operation sends the reconfiguration request to the containing assembly, which performs the reconfiguration if it approves the request. The class `ConnectionRequest` is also provided by the JAVA/A framework; its constructor has parameters which indicate the originating component of the request, both affected components and their ports, and the connector to use. If the request is not approved by the containing assembly, a reconfiguration exception is thrown. The other reconfiguration variants are realised in JAVA/A following a similar scheme.

3 A Semantical Model for Architectural Programming

We present a semantical model for the main constituents of our component metamodel of JAVA/A (see Fig. 1). The semantical model uses a states-as-algebras approach [4] for representing the internals of components and assemblies, and the interface automata approach [2] to I/O-transition systems for describing the observable behaviour. Algebraic operation interfaces specify which operations are provided and required by a port. Port protocols are given by I/O-transition systems with input and output labels over the operations the port provides and requires. Note that the semantics does not distinguish between simple components and composite components. The semantics of both, a simple component and a composite component, is a component which is defined through its internal, algebraic state space and the declaration of the ports it offers; I/O-transition systems with input and output labels over the port operations describe the possible component behaviours. The semantics of a composite component is given by first taking the semantics of its assembly and then by hiding the internal structure of how that assembly is built. Assemblies are given by an internal, algebraic state space and a declaration of the components and connectors between component ports which may occur; the assembly behaviour is again described by an I/O-transition system whose labels reflect the observations on component ports that are not connected and the synchronisation of connected component ports. The internal state of a component stores which ports it currently offers thus providing means for changing ports dynamically; similarly, the internal state of an assembly records the current components and connectors, which may vary over time.

3.1 Preliminaries

We first summarise some basic definitions on algebras and I/O-transition systems. In particular, we recall the notion of algebraic signatures, algebras, and reducts, see [19]. For I/O-transition systems [2], we define relabellings and products.

Algebras

A *signature* $\Sigma = (S, F)$ consists of a set S of *sorts* and a set F of *function symbols* $f : s_1, \ldots, s_n \to s$. A signature $\Sigma = (S, F)$ is a *subsignature* of a signature $\Sigma' = (S', F')$, denoted by $\Sigma \subseteq \Sigma'$, if $S \subseteq S'$ and $F \subseteq F'$. A *(total)* Σ*-algebra* $A = ((s^A)_{s \in S}, (f^A)_{f \in F})$ consists of a set s^A for each sort s and interpretation functions $f^A : s_1^A, \ldots, s_n^A \to s^A$ for each $f : s_1, \ldots, s_n \to s \in F$. For signatures $\Sigma = (S, F)$, $\Sigma' = (S', F')$ with $\Sigma \subseteq \Sigma'$, the Σ*-reduct* of a Σ'-algebra A is given by $A|_\Sigma = ((s^A)_{s \in S}, (f^A)_{f \in F})$. If A is a Σ-algebra and $X = (x_i : s_i)_{i \in I}$ is an S-sorted set of variables, a *valuation* $\rho : X \to A$ assigns to each variable x of sort $s \in S$ a value in s^A.

I/O-transition systems

An *I/O-labelling* (I, O, T) consists of three mutually disjoint sets of *input* (or *provided*) labels I, *output* (or *required*) labels O, and *internal* labels T. An *I/O-*

transition system $A = (Q, B, \Delta)$ over an I/O-labelling (I, O, T) is given by a set of *states* Q, a set of *initial states* $B \subseteq Q$ and a *transition relation* $\Delta \subseteq Q \times (I \cup O \cup T) \times Q$.

An *I/O-relabelling* $\lambda : L_1 \to L_2$ from an I/O-labelling $L_1 = (I_1, O_1, T_1)$ to an I/O-labelling $L_2 = (I_2, O_2, T_2)$ consists of three functions $\lambda_I : I_1 \to I_2$, $\lambda_O : O_1 \to O_2$, and $\lambda_T : T_1 \to T_2$. For a label $l \in I_1 \cup O_1 \cup T_1$, we write $\lambda(l)$ for $\lambda_I(l)$, if $l \in I_1$; $\lambda_O(l)$, if $l \in O_1$; and $\lambda_T(l)$, if $l \in T_1$. Let $A = (Q, B, \Delta)$ be an I/O-transition system over L and let $\lambda : L \to L'$ be an I/O-relabelling. Then $A' = (Q', B', \Delta')$ defined by $Q' = Q$, $B' = B$, $\Delta' = \{(q, \lambda(l), q') \mid (q, l, q') \in \Delta\}$ is the *relabelled* I/O-transition system over L' with respect to λ, written as $A\lambda$.

Two I/O-labellings $L_1 = (I_1, O_1, T_1)$ and $L_2 = (I_2, O_2, T_2)$ are *composable* if $T_1 \cap (I_2 \cup O_2 \cup T_2) = \emptyset$, $I_1 \cap I_2 = \emptyset$, $O_1 \cap O_2 = \emptyset$, $T_2 \cap (I_1 \cup O_1 \cup T_1) = \emptyset$. The *shared* labels of L_1 and L_2, written $L_1 \cap L_2$, are given by $(I_1 \cup O_1 \cup T_1) \cap (I_2 \cup O_2 \cup T_2)$. The *product* I/O-labelling of L_1 and L_2, written as $L_1 \otimes L_2$ is given by $((I_1 \cup I_2) \setminus (L_1 \cap L_2), (O_1 \cup O_2) \setminus (L_1 \cap L_2), T_1 \cup T_2 \cup (L_1 \cap L_2))$. Let $A_1 = (Q_1, B_1, \Delta_1)$ and $A_2 = (Q_2, B_2, \Delta_2)$ be I/O-transition systems over composable I/O-labellings L_1 L_2, respectively. The *product* of A_1 and A_2, written as $A_1 \otimes A_2$, is given by the I/O-transition system (Q, B, Δ) over the product I/O-labelling $L_1 \otimes L_2$ defined as follows:

(i) $Q = Q_1 \times Q_2$;

(ii) $B = B_1 \times B_2$;

(iii) $\Delta = \{((q_1, q_2), l, (q_1', q_2)) \mid (q_1, l, q_1') \in \Delta_1 \wedge l \notin L_1 \cap L_2 \wedge q_2 \in Q_2\} \cup$
$\{((q_1, q_2), l, (q_1, q_2')) \mid (q_2, l, q_2') \in \Delta_1 \wedge l \notin L_1 \cap L_2 \wedge q_1 \in Q_1\} \cup$
$\{((q_1, q_2), l, (q_1', q_2')) \mid (q_1, l, q_1') \in \Delta_1 \wedge (q_2, l, q_2') \in \Delta_2 \wedge l \in L_1 \cap L_2\}$.

3.2 Interfaces and ports

Interfaces describe a set of accepted messages by declaring operations over a data signature. Formally, an *interface* is given by a pair (Σ, Op) of a signature Σ and a set of operations Op over Σ, where an *operation* $op \in Op$ takes the form $nm(x_1 : s_1, \ldots, x_k : s_k)$ with nm the operation name and $\mathrm{var}(op) = x_1 : s_1, \ldots, x_k : s_k$ the typed formal parameters with sorts in Σ. A *message* for an operation op over Σ is a pair (op, ρ) with $\rho : \mathrm{var}(op) \to \sigma$ a valuation from the operation's formal parameters to a Σ-algebra σ. The class of messages for a set of operations Op over a signature Σ is denoted by $Msg_\Sigma(Op)$.

For instance, the provided interface of port BA of component Bank in our running example has the signature $(S_{\mathsf{BA}}, F_{\mathsf{BA}})$ with sorts $S_{\mathsf{BA}} = \{\mathsf{int}, \mathsf{IBAN}, \mathsf{Money}\}$ and F_{BA} comprising basic functions on integers, amounts of money, &c.; the declared operations are verifyPIN(iban : IBAN, pin : int) and withdraw(iban : IBAN, amount : Money).

Ports represent the windows of a component, prescribing which messages are accepted at a port by specifying a provided interface and which messages must be understood from a port by specifying a required interface. Thus we define a *port signature* Σ_P to be a pair (I, O) consisting of a *provided* interface $I = (\Sigma_P^{\mathrm{pro}}, Op_P^{\mathrm{pro}})$ and a *required* interface $O = (\Sigma_P^{\mathrm{req}}, Op_P^{\mathrm{req}})$.

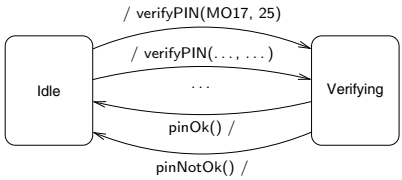


Figure 5. I/O-transition system for port BC.

A port protocol regulates the order and dependencies of messages from and to a port. Such a port protocol for a port signature Σ_P is given by a *port I/O-transition system* (Q, B, Δ) over the I/O-labelling $(Label_P^{\text{pro}}, Label_P^{\text{req}}, Label_P^{\text{int}})$ where the labels are either

(i) an *input* label $i/ \in Label_P^{\text{pro}}$ with $i = (op, \rho) \in Msg(Op_P^{\text{pro}})$; or

(ii) an *output* label $/o \in Label_P^{\text{req}}$ with $o = (op, \rho) \in Msg(Op_P^{\text{req}})$; or

(iii) the *internal* label $/ \in Label_P^{\text{int}}$.

A *port declaration* $P : \Sigma_P$ consists of a *name* P and a port signature Σ_P.

Figure 5 shows an excerpt of a possible I/O-transition system for port BC of component Bank which reflects the UML state machine specifying the port protocol in Fig. 3.

3.3 Components

A component has an internal data state and declares ports. Formally, a *component signature* Σ_C is a pair $(\Sigma, (P : \Sigma_P)_{P \in \mathscr{P}})$ where Σ is the *state* signature of Σ_C and $(P : \Sigma_P)_{P \in \mathscr{P}}$ the family of *port declarations* of Σ_C satisfying the following requirements:

(i) all $P \in \mathscr{P}$ are sorts of Σ;

(ii) $\Sigma_P^{\text{pro}}, \Sigma_P^{\text{req}} \subseteq \Sigma$ for all $P \in \mathscr{P}$.

Note that the port names used in the port declarations are required to be sorts in the state signatures. This allows for dynamic ports, where each port is represented by an element of appropriate sort given by the port declarations. We write Σ_C^{state} for the state signature of the component signature Σ_C; and $Ports_C$ for the names of the port declarations $\mathscr{P}$ of Σ_C. A *component declaration* $C : \Sigma_C$ consists of a *name* C and a component signature Σ_C.

The behaviour of a component is given by an I/O-transition system. The states of the transition system have the function of control states (cf. [6]) as they determine the behaviour of the component. A state operator maps each control state to a data state which is an algebra over the state signature of the component. The labels of the transitions are either the internal label or I/O-labels corresponding to the messages sent and received via ports.

The *state space* $State_C$ of a component signature Σ_C is given by $\text{Alg}(\Sigma_C^{\text{state}})$. A *label* for a component signature Σ_C and a state pair $(\sigma, \sigma') \in State_C \times State_C$ is either

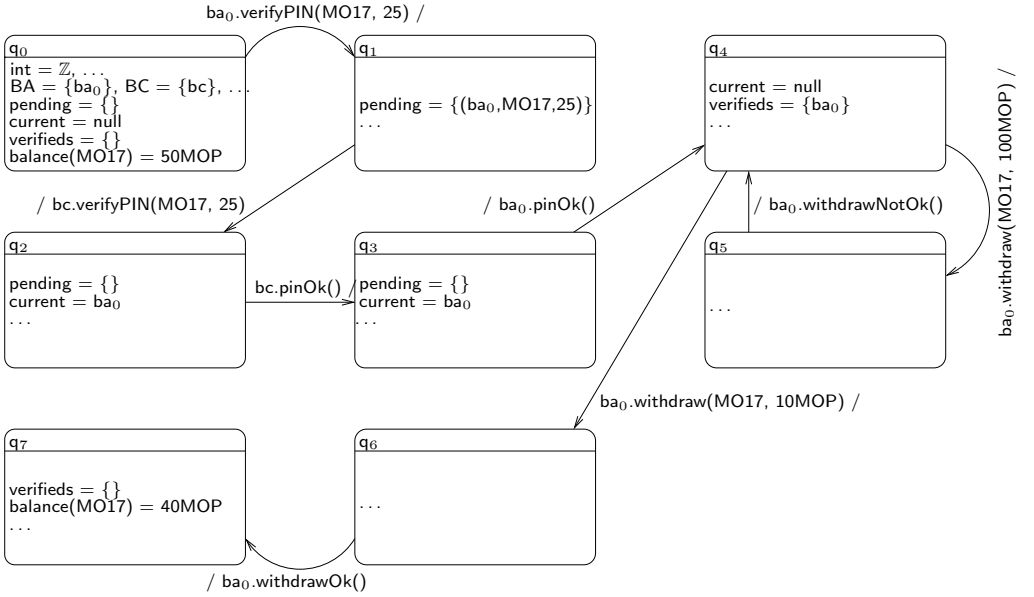


Figure 6. A model for the component Bank.

(i) an *input* label $p.i/ \in Label_C^{\mathrm{pro}}(\sigma,\sigma')$ with $p \in P^\sigma$ for some $P \in Ports_C$ and $i = (op,\rho) \in Msg(Op_P^{\mathrm{pro}})$ where $\rho : \mathrm{var}(op) \to \sigma' | \Sigma_P^{\mathrm{pro}}$; or

(ii) an *output* label $/p.o \in Label_C^{\mathrm{req}}(\sigma,\sigma')$ with $p \in P^\sigma$ for some $P \in Ports_C$ and $o = (op,\rho) \in Msg(Op_P^{\mathrm{req}})$ where $\rho : \mathrm{var}(op) \to \sigma | \Sigma_P^{\mathrm{req}}$; or

(iii) the *internal* label $/ \in Label_C^{\mathrm{int}}(\sigma,\sigma')$.

We define $Label_C(\sigma,\sigma')$ as $Label_C^{\mathrm{pro}}(\sigma,\sigma') \cup Label_C^{\mathrm{req}}(\sigma,\sigma') \cup Label_C^{\mathrm{int}}(\sigma,\sigma')$; and we abbreviate $\bigcup\{Label_C^{\mathrm{pro}}(\sigma,\sigma') \mid (\sigma,\sigma') \in State_C \times State_C\}$ by $Label_C^{\mathrm{pro}}$, and similarly for $Label_C^{\mathrm{req}}$ and $Label_C^{\mathrm{int}}$.

An *I/O-transition system* for a component signature Σ_C is given by an I/O-transition system (Q,B,Δ) over the I/O-labelling $(Label_C^{\mathrm{pro}}, Label_C^{\mathrm{req}}, Label_C^{\mathrm{int}})$. A *state operator* for a component signature Σ_C and an I/O-transition system (Q,B,Δ) for Σ_C is a function $\varsigma : Q \to State_C$ such that $l \in Label_C(\varsigma(q),\varsigma(q'))$, if $(q,l,q') \in \Delta$. A *model* of a component signature Σ_C is given by a pair (H,ς) such that H is an I/O-transition system for Σ_C and ς is state operator for Σ_C and H. The class of models of Σ_C is denoted by $\mathrm{Mod}(\Sigma_C)$.

An example of a model for the component Bank is given in Fig. 6. The control states, q_0, q_1, ..., are shown in the upper part of the state boxes and the associated data states in the lower part.

From an I/O-transition system $H = (Q,B,\Delta)$ of a component it is possible to generate the behaviour of the component observable at a given port p of sort $P \in Ports_C$ as an I/O-transition system over the signature of that port. The states of this reduct of H to p are given by all states of the component in which the port lives. The transitions of the reduct are the transitions from the component where the prefix $p.$ in the labels involving port p is removed and all other labels are converted to internal transitions. The start states include all the start states of the

component H if port p exists in these states, as well as the states where p starts to exist.

(i) $Q\restriction_\varsigma p = \{q \mid q \in Q \wedge p \in P^{\varsigma(q)}\}$;

(ii) $B\restriction_\varsigma p = \{q \in Q \mid (q \in B \cap Q\restriction_\varsigma p) \vee (\exists (q', l, q) \in \Delta \,.\, p \notin P^{\varsigma(q')} \wedge p \in P^{\varsigma(q)})\}$;

(iii) $\Delta\restriction_\varsigma p = \{(q, l\restriction p, q') \mid (q, l, q') \in \Delta \cap (Q\restriction_\varsigma p) \times Label_C \times (Q\restriction_\varsigma p)\}$ where $(p.i/)\restriction p = i/$, $(/p.o)\restriction p = /o$, and $l\restriction p = /$ otherwise.

The *reduct* of H to p via ς, denoted by $H\restriction_\varsigma p$, is the I/O-transition system $(Q\restriction_\varsigma p, B\restriction_\varsigma p, \Delta\restriction_\varsigma p)$ over the I/O-labelling $(Label_P^{\text{pro}}, Label_P^{\text{req}}, Label_P^{\text{int}})$. For a model $M = ((Q, B, \Delta), \varsigma) \in \text{Mod}(\Sigma_C)$, a $P \in Ports_C$, a $p \in \bigcup\{P^{\varsigma(q)} \mid q \in Q\}$ we write $M\restriction p$ for $(Q, B, \Delta)\restriction_\varsigma p$.

3.4 Connectors

A connector connects two ports. The signatures of the ports connected by a connector are given by the connector's signature. These port signatures need to be compatible, i.e., the provided interface of each port needs to subsume the required interface of the other port.

A *connector signature* Σ_A is a pair $(P_1 : \Sigma_{P_1}, P_2 : \Sigma_{P_2})$, where $\Sigma_{P_1} = (I_1, O_1)$ and $\Sigma_{P_2} = (I_2, O_2)$ are the *left* and *right* port signatures of Σ_A such that I_2 subsumes O_1 and I_1 subsumes O_2 where an interface $I' = (\Sigma', Op')$ *subsumes* an interface $I = (\Sigma, Op)$, written as $I \preceq I'$, if $\Sigma \subseteq \Sigma'$ and $Op \subseteq Op'$. A *connector declaration* $A : \Sigma_A$ consists of a name A and a connector signature Σ_A. We write $Ports_A$ for the set $\{P_1, P_2\}$ with the port names used in the connector declaration.

3.5 Assemblies

Additional to comprising component and connector declarations, an assembly has an algebraic state signature such that the names of components used in the component declarations and the names of the connectors used in the connector declarations correspond to sorts in the state signature. Then components and connectors are represented by elements of these sorts. Similarly to ports, this allows for adding and removing components and connectors to the assembly at runtime.

An *assembly signature* Σ_Γ is a triple $(\Sigma, (C : \Sigma_C)_{C \in \mathscr{C}}, (A : \Sigma_A)_{A \in \mathscr{A}})$ where Σ is the *state* signature of Σ_Γ, $(C : \Sigma_C)_{C \in \mathscr{C}}$ is the family of *component declarations* of Σ_Γ and $(A : \Sigma_A)_{A \in \mathscr{A}}$ is the family of *connector declarations* of Σ_Γ satisfying the following requirements:

(i) All $C \in \mathscr{C}$ are sorts of Σ;

(ii) for all $C \in \mathscr{C}$ all port names $P \in Ports_C$ are sorts in Σ;

(iii) all $A \in \mathscr{A}$ are sorts of Σ;

(iv) for all $A \in \mathscr{A}$ with $\Sigma_A = (P_1 : \Sigma_{P_1}, P_2 : \Sigma_{P_2})$ there are $C_1, C_2 \in \mathscr{C}$ such that $P_1 \in Ports_{C_1}$ and $P_2 \in Ports_{C_2}$.

(v) for all $A \in \mathscr{A}$ with $\Sigma_A = (P_1 : \Sigma_{P_1}, P_2 : \Sigma_{P_2})$ there are function symbols $\pi_{1,A} : A \to P_1$ and $\pi_{2,A} : A \to P_2$ in Σ;

We write $\Sigma_\Gamma^{\text{state}}$ for the state signature of the assembly signature Σ_Γ; $Comp_\Gamma$ for the names of the component declarations $\mathscr{C}$ of Σ_Γ; and $Conn_\Gamma$ for the names of the connector declarations $\mathscr{A}$ of Σ_Γ.

As with ports and components, the behaviour of an assembly is again an I/O-transition system (Q, B, Δ). As with components, there is a state operator mapping each state in Q to an algebra over the state signature of the assembly. In addition, there is a component reduct functor which, given a component c, maps an algebra over the state signature of the assembly to an algebra over the state signature of the component c. The labels of the transition system have the form $/$, $c.p.i/$, $/c.p.o$, and $(c_1.p_1, c_2.p_2).m$, corresponding to the internal label, to messages i sent by component c via port p, to messages o received by component c via port p, and to messages m, where the sending of message m by component c_1 via port p_1 and the reception of that message by component c_2 via port p_2 are synchronised, respectively. Synchronisation labels can only occur, and are bound to occur, if compatible ports of components are linked by a connector.

Let Σ_Γ be an assembly signature. The *state space* $State_\Gamma$ for Σ_Γ is given by $\text{Alg}(\Sigma_\Gamma^{\text{state}})$. An element of $State_\Gamma$ is called a *configuration*. A *component reduct operator* for Σ_Γ is a function $\varrho \in \Pi\sigma \in State_\Gamma \,.\, \Pi C \in Comp_\Gamma \,.\, \Pi c \in C^\sigma \,.\, State_C$. A *label* for an assembly signature Σ_Γ, a component reduct operator ϱ, and a state pair $(\sigma, \sigma') \in State_\Gamma \times State_\Gamma$ is either

(i) an *input* label $c.p.i/ \in Label^{\text{pro}}_{\Gamma,\varrho}(\sigma, \sigma')$ with $c \in C^\sigma$ for some $C \in Comp_\Gamma$, $p \in P^\sigma$ for some $P \in Ports_C$ and $i = (op, \rho) \in Msg(Op^{\text{pro}}_P)$ with $\rho : \text{var}(op) \to (\varrho(\sigma')(C)(c))\restriction\Sigma^{\text{pro}}_P$ if there is no $A \in Conn_\Gamma$ such that there is an $a \in A^\sigma$ with $p \in \{\pi^\sigma_{1,A}(a), \pi^\sigma_{2,A}(a)\}$; or

(ii) an *output* label $/c.p.o \in Label^{\text{req}}_{\Gamma,\varrho}(\sigma, \sigma')$ with $c \in C^\sigma$ for some $C \in Comp_\Gamma$, $p \in P^\sigma$ for some $P \in Ports_C$ and $o = (op, \rho) \in Msg(Op^{\text{req}}_P)$ with $\rho : \text{var}(op) \to (\varrho(\sigma)(C)(c))\restriction\Sigma^{\text{req}}_P$ if there is no $A \in Conn_\Gamma$ such that there is an $a \in A^\sigma$ with $p \in \{\pi^\sigma_{1,A}(a), \pi^\sigma_{2,A}(a)\}$; or

(iii) a *synchronisation* label $(c_1.p_1, c_2.p_2).m \in Label^{\text{int}}_{\Gamma,\varrho}(\sigma, \sigma')$ with $c_1 \in C_1^\sigma$ for some $C_1 \in Comp_\Gamma$, $p_1 \in P_1^\sigma$ for some $P_1 \in Ports_{C_1}$, $c_2 \in C_2^\sigma$ for some $C_2 \in Comp_\Gamma$, $p_2 \in P_2^\sigma$ for some $P_2 \in Ports_{C_2}$ such that $m = (op, \rho) \in Msg(Op^{\text{pro}}_{P_1}) \cap Msg(Op^{\text{req}}_{P_2})$ with $\rho : \text{var}(op) \to (\varrho(\sigma)(C_1)(c_1))\restriction(\Sigma^{\text{pro}}_{P_1} \cap \Sigma^{\text{req}}_{P_2}) = (\varrho(\sigma')(C_2)(c_2))\restriction(\Sigma^{\text{req}}_{P_2} \cap \Sigma^{\text{pro}}_{P_1})$ if there is an $A \in Conn_\Gamma$ and an $a \in A^\sigma$ with $p_1 = \pi^\sigma_{1,A}(a)$ and $p_2 = \pi^\sigma_{2,A}(a)$; or

(iv) the *internal* label $/ \in Label^{\text{int}}_{\Gamma,\varrho}(\sigma, \sigma')$.

We define $Label_{\Gamma,\varrho}(\sigma, \sigma')$ as $Label^{\text{pro}}_{\Gamma,\varrho}(\sigma, \sigma') \cup Label^{\text{req}}_{\Gamma,\varrho}(\sigma, \sigma') \cup Label^{\text{int}}_{\Gamma,\varrho}(\sigma, \sigma')$, and we abbreviate $\bigcup\{Label^{\text{pro}}_{\Gamma,\varrho}(\sigma, \sigma') \mid (\sigma, \sigma') \in State_\Gamma \times State_\Gamma\}$ by $Label^{\text{pro}}_{\Gamma,\varrho}$, and similarly for $Label^{\text{req}}_{\Gamma,\varrho}$ and $Label^{\text{int}}_{\Gamma,\varrho}$.

An *I/O-transition system* for an assembly signature Σ_Γ and a component reduct operator ϱ for Σ_Γ is given by an I/O-transition system (Q, B, Δ) over the I/O-labelling $(Label^{\text{pro}}_{\Gamma,\varrho}, Label^{\text{req}}_{\Gamma,\varrho}, Label^{\text{int}}_{\Gamma,\varrho})$. A *state operator* for a component signature Σ_C and an I/O-transition system (Q, B, Δ) for Σ_Γ and a component reduct operator

ϱ for Σ_Γ is a function $\varsigma : Q \to State_\Gamma$ such that $l \in Label_{\Gamma,\varrho}(\varsigma(q), \varsigma(q'))$, if $(q, l, q') \in \Delta$.

Similarly to the reduct of components to ports in Sect. 3.3, we define the reduct of a I/O-transition system H of an assembly to a I/O-transition system representing a component c within the assembly. The states of the reduct are given by all states of H where the component lives, and the transitions of the reduct are the transitions of H where the prefix $c.$ in the labels involving component c are removed. Furthermore, a synchronisation label of the form $(c_1.p_1, c_2.p_2).m$ is reduced to $/p_1.m$ if $c_1 = c$ and to $p_2.m/$ if $c_2 = c$ and to $/$ if neither $c_1 = c$ nor $c_2 = c$. All other labels are mapped to the internal label $/$. The start states are all the start states of H together with all the states in which the component c starts to exist.

Let $H = (Q, B, \Delta)$ be an I/O-transition system for an assembly signature Σ_Γ and a component reduct operator ϱ and let ς be a state operator for Σ_C, H, and ϱ. Let $C \in Comp_\Gamma$ and $c \in \bigcup\{C^{\varsigma(q)} \mid q \in Q\}$. The *reduct* of H to c via ς, denoted by $H\restriction_\varsigma c$, is the I/O-transition system $(Q\restriction_\varsigma c, B\restriction_\varsigma c, \Delta\restriction_\varsigma c)$ over the I/O-labelling $(Label_C^{\text{pro}}, Label_C^{\text{req}}, Label_C^{\text{int}})$ with

(i) $Q\restriction_\varsigma c = \{q \mid q \in Q \wedge c \in C^{\varsigma(q)}\}$;

(ii) $B\restriction_\varsigma c = \{q \in Q \mid (q \in B \cap Q\restriction_\varsigma c) \vee (\exists (q', l, q) \in \Delta \,.\, c \notin C^{\varsigma(q')} \wedge c \in C^{\varsigma(q)})\}$;

(iii) $\Delta\restriction_\varsigma c = \{(q, l\restriction c, q') \mid (q, l, q') \in \Delta \cap (Q\restriction_\varsigma c) \times Label_\Gamma \times (Q\restriction_\varsigma c)\}$ where $(c.p.i/)\restriction c = p.i/$, $(/c.p.o)\restriction c = /p.o$, $(c_1.p_1, c_2.p_2).m\restriction c = p_1.m/$ if $c = c_1$, $(c_1.p_1, c_2.p_2).m\restriction c = /p_2.m$ if $c = c_2$, and $l\restriction c = /$ otherwise.

The *reduct* of ϱ to c via ς, denoted by $\varrho\restriction_\varsigma c$, is given by $\lambda q \,.\, \varrho(\varsigma(q))(C)(c)$.

A model of an assembly signature is given by an I/O-transition system H plus a state operator, mapping a state of the transition system to an algebra over the state signature of the assembly, and a component reduct operator, extracting an algebra over the state signature of a component from an algebra over the state signature of the assembly. Each reduct of the assembly to a component is required to be a model of the corresponding component signature. Furthermore, existing ports in a data state of the assembly have to also exist in the data state of exactly one component of the assembly.

A model for the assembly Bank–ATM is shown in Fig. 7. As with component models, the control states, q_0, q_1, ..., are shown in the upper part of the state boxes, the associated data states in the lower part. The data states are parameterised over the component instances, such that $\mathsf{current(b)} = \mathsf{null}$ represents the valuation of the attribute current of Bank instance b.

A *model* of an assembly signature Σ_Γ is given by a triple (ϱ, H, ς) where ϱ is a component reduct operator Σ_Γ, $H = (Q, B, \Delta)$ is an I/O-transition system for Σ_Γ and ϱ, and ς is a state operator for Σ_Γ, H, and ϱ such that

(i) for all $C \in \mathscr{C}$ and all $c \in \bigcup\{C^{\varsigma(q)} \mid q \in Q_\Gamma \wedge C \in Comp_\Gamma\}$ the following holds: $(H\restriction_\varsigma c, \varrho\restriction_\varsigma c) \in \text{Mod}(\Sigma_C)$;

(ii) if $q \in Q$ with $p \in P^{\varsigma(q)}$ for some $P \in Ports_C$ for some $C \in Comp_\Gamma$, then there is a unique $c \in C^{\varsigma(q)}$ such that $p \in P^{\varrho(\varsigma(q))(C)(c)}$;

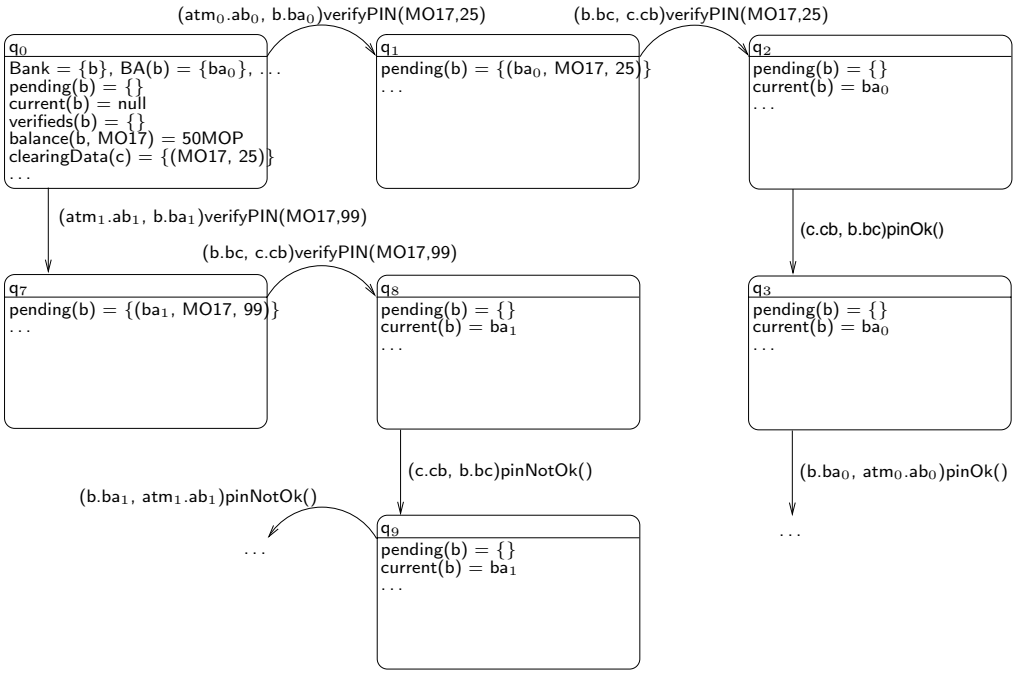


Figure 7. A model for the assembly Bank–ATM.

(iii) if $q \in Q$ with $p \in P^{\varrho(\varsigma(q))(C)(c)}$ for some $P \in \mathit{Ports}_C$ for some $C \in \mathit{Comp}_\Gamma$ and some $c \in C^{\varsigma(q)}$, then $p \in P^{\varsigma(q)}$.

The class of models of Σ_Γ is denoted by $\mathrm{Mod}(\Sigma_\Gamma)$. For a model $M = (\varrho, (Q, B, \Delta), \varsigma) \in \mathrm{Mod}(\Sigma_\Gamma)$, a $C \in \mathit{Comp}_\Gamma$ and a $c \in \bigcup\{C^{\varsigma(q)} \mid q \in Q\}$ we write $M{\restriction}c$ for $((Q, B, \Delta){\restriction}_\varsigma c, \varrho{\restriction}_\varsigma c)$.

3.6 From assemblies to components

As shown in Fig. 1, the semantics of a composite component is given by its assembly by hiding the internal structure of the assembly, i.e., its subcomponents and connectors. The state signature of the component is given by the state signature of the assembly, and the ports of the component are all the ports of the assembly which are not connected within that assembly. Let $\Sigma_\Gamma = (\Sigma, (C : \Sigma_C)_{C \in \mathscr{C}}, (A : \Sigma_A)_{A \in \mathscr{A}})$ be an assembly signature, then the corresponding component signature $\mathit{BuildComp}(\Sigma_\Gamma)$ is $(\Sigma, (P : \Sigma_P)_{P \in \mathscr{P}})$ where $(P : \Sigma_P)$ belongs to $(P : \Sigma_P)_{P \in \mathscr{P}}$ if $(P : \Sigma_P)$ is a port declaration in the component signature Σ_C for some component declaration $(C : \Sigma_C)$ of Σ_Γ such that P is not a port name in $\bigcup_{A \in \mathscr{A}} \mathit{Ports}_A$.

The I/O-transition system of the component determined by an assembly is given by the transition system for the assembly with labels properly renamed, i.e., the component prefix $c.$ is removed from I/O labels and all other labels (i.e., synchronisation labels and the internal label) are mapped to the internal label /.

Let (ϱ, H, ς) be a model of an assembly signature Σ_Γ and H an I/O-transition system for Σ_Γ. Then $\mathit{BuildComp}((\varrho, H, \varsigma)) = (H\lambda, \varsigma)$ is a model of component signature $\mathit{BuildComp}(\Sigma_\Gamma)$ where λ is an I/O-relabelling given by $\lambda(c.p.i/) = p.i/$,

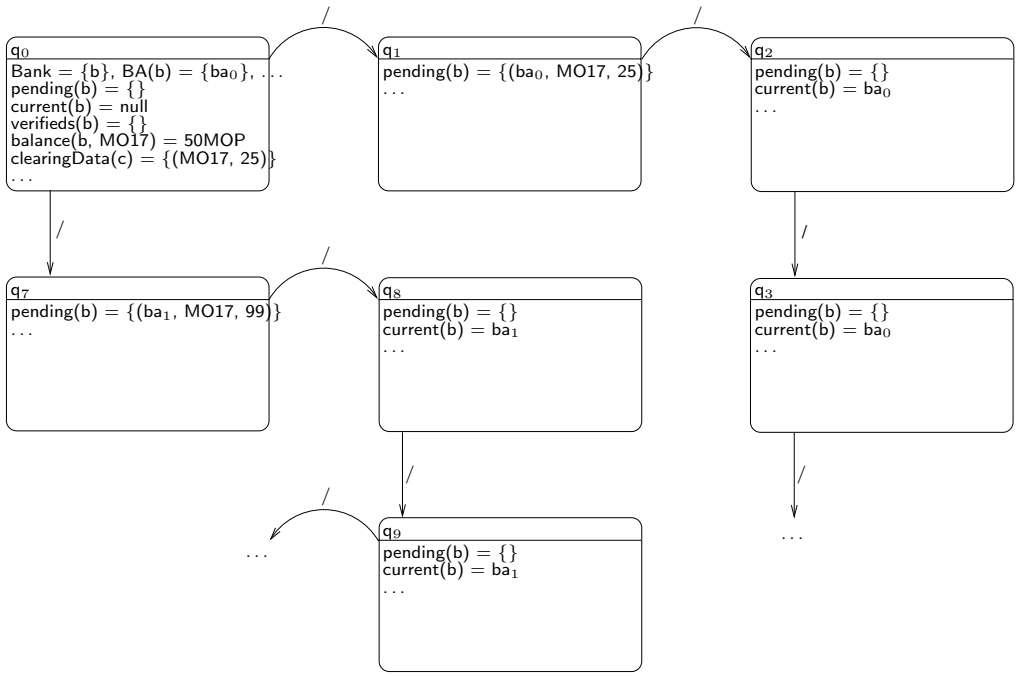


Figure 8. A model for the component built from the assembly Bank–ATM of Fig. 7.

$\lambda(/c.p.o) = /p.o$, $\lambda((c_1.p_1, c_2.p_2).m) = /$, and $\lambda(/) = /$.

Figure 8 shows the component Bank–ATM based on the assembly Bank–ATM from Fig. 7. Again the control states, q_0, q_1, ..., are shown in the upper part of the state boxes and the associated data states in the lower part. Since the assembly Bank–ATM does not have any open ports, the model in Fig. 8 has only internal labels and no I/O-labels. The synchronisation labels of the assembly Bank–ATM (shown in Fig. 7) have been converted to the internal label in the component Bank–ATM.

4 Deadlock-free components

Based on the semantical component model, we prove that model checking port protocols, as described in Sect. 2.1, leads to correct results: If the protocols are compatible, i.e., the provided interface of one port subsumes the required interface of the other port and vice versa, and the product of the two port protocol machines is deadlock-free, then a component built from an assembly is deadlock-free provided that the components of the assembly refine the respective port protocol state machines.

4.1 Refinement

We first introduce a special notion of refinement of I/O-transition systems which preserves deadlock-freedom and extends the definition of alternating simulation refinement [2].

Let $A = (Q, B, \Delta)$ be an I/O-transition system over (I, O, T) and $q \in Q$. The *internal closure* of q, written $\text{closure}_A(q)$ is the smallest set $E \subseteq Q$ such that (1) $q \in E$ and (2) if $q' \in E$ and $(q', l, q'') \in \Delta$ with $l \in T$ then $q'' \in E$. The *enabled labels* of q are given by the set $\text{enabled}_A(q) = \{l \in I \cup O \mid \exists q' \in \text{closure}_A(q) \,.\, \exists q'' \in Q \,.\, (q', l, q'') \in \Delta\}$. The states *reachable* by an enabled label l in q are given by the set $\text{reach}_A(q, l) = \{q'' \mid \exists (q', l, q'') \in \Delta \,.\, q' \in \text{closure}_A(q)\}$; the states *reachable* by enabled transitions from q are given by the set $\text{reach}_A(q) = \bigcup\{\text{reach}_A(q, l) \mid l \in \text{enabled}_A(q)\}$. The *dead* states reachable by internal labels from q are given by the set $\text{dead}_A(q) = \{q' \in \text{closure}_A(q) \mid \neg\exists (q', l, q'') \in \Delta\}$.

Let $A_1 = (Q_1, B_1, \Delta_1)$ and $A_2 = (Q_2, B_2, \Delta_2)$ be I/O-transition systems. A relation $R \subseteq Q_1 \times Q_2$ is a *simulation* from A_2 to A_1 if the following conditions hold for all $(q_1, q_2) \in R$:

(i) $\text{enabled}_{A_2}(q_2) = \text{enabled}_{A_1}(q_1)$;

(ii) for all $l \in \text{enabled}_{A_2}(q_2)$ and $q_2' \in \text{reach}_{A_2}(q_2, l)$, there is a $q_1' \in \text{reach}_{A_1}(q_1, l)$ such that $(q_1', q_2') \in R$;

(iii) if $\text{dead}_{A_2}(q_2) \neq \emptyset$ then $\text{dead}_{A_1}(q_1) \neq \emptyset$.

An I/O-transition system $A_2 = (Q_2, B_2, \Delta_2)$ over I/O-labelling (I_2, O_2, T_2) *refines* an I/O-transition system $A_1 = (Q_1, B_1, \Delta_1)$ over I/O-labelling (I_1, O_1, T_1), written $A_1 \succeq A_2$, if the following conditions hold:

(i) $I_1 \subseteq I_2$ and $O_2 \subseteq O_1$;

(ii) there is a simulation R from A_2 to A_1 such that for all $q_{0,2} \in B_2$ there is a $q_{0,1} \in B_1$ with $(q_{0,1}, q_{0,2}) \in R$.

An I/O-transition system $A = (Q, B, \Delta)$ has a *deadlock*, written as $A \models \delta$, if there is a sequence $q_0, q_1, \ldots, q_n \in Q$ such that $q_0 \in B$ and $q_{i+1} \in \text{reach}_A(q_i)$ for all $0 \leq i < n$ and $\text{dead}_A(q_n) \neq \emptyset$. A is *deadlock-free*, if $A \not\models \delta$.

Lemma 4.1 *Let A_1 and A_2 be I/O-transition systems. If $A_1 \not\models \delta$ and $A_1 \succeq A_2$, then $A_2 \not\models \delta$.*

Proof By contradiction, simulating a deadlocking run of A_2 in A_1 and using (iii).□

4.2 Deadlock-free assemblies

We now consider components which are correct in the sense that they refine their respective port protocol state machines. We prove that correct components connected via compatible ports lead to deadlock-free assemblies for the special case that the assembly has exactly two components and each of the components has one port which is connected to the port of the other component.

Let Σ_{P_l} and Σ_{P_r} be port signatures, let $\Sigma_{C_\mathrm{l}} = (\Sigma_\mathrm{l}, \{P_\mathrm{l} : \Sigma_{P_\mathrm{l}}\})$ and $\Sigma_{C_\mathrm{r}} = (\Sigma_\mathrm{r}, \{P_\mathrm{r} : \Sigma_{P_\mathrm{r}}\})$ be component signatures, let $\Sigma_A = (\Sigma_{P_\mathrm{l}}, \Sigma_{P_\mathrm{r}})$ be a connector signature, and let $\Sigma_\Gamma = (\Sigma, \{C_\mathrm{l} : \Sigma_{C_\mathrm{l}}, C_\mathrm{r} : \Sigma_{C_\mathrm{r}}\}, \{A : \Sigma_A\})$ be an assembly signature. Let $M_\Gamma = (\varrho_\Gamma, H_\Gamma, \varsigma_\Gamma) \in \text{Mod}(\Sigma_\Gamma)$ with $H_\Gamma = (Q_\Gamma, B_\Gamma, \Delta_\Gamma)$ such that

(i) there are always exactly two components, i.e., there are $c_\mathrm{l}, c_\mathrm{r}$ with $\{c_\mathrm{l}\} = C_\mathrm{l}^{\varsigma_\Gamma(q)}$

and $\{c_r\} = C_r^{\varsigma_\Gamma(q)}$ for all $q \in Q_\Gamma$;

(ii) each existing component has always exactly one port, i.e., there are p_l, p_r with $\{p_l\} = P_l^{\varrho_\Gamma(\varsigma_\Gamma(q))(C_l)(c_l)}$ and $\{p_r\} = P_r^{\varrho_\Gamma(\varsigma_\Gamma(q))(C_r)(c_r)}$ for all $q \in Q_\Gamma$;

(iii) the ports are always connected, i.e., there is an a with $\{a\} = A^{\varsigma_\Gamma(q)}$ and $\pi_{1,A}^{\varsigma_\Gamma(q)}(a) = p_l$ and $\pi_{2,A}^{\varsigma_\Gamma(q)}(a) = p_r$ for all $q \in Q_\Gamma$.

Let $M_l = (Q_l, B_l, \Delta_l)$ and $M_r = (Q_r, B_r, \Delta_r)$ be I/O-transition systems for Σ_{P_l} and Σ_{P_r}, respectively. Let $M_l' = M_l\lambda_l$ where $\lambda_l(/) = p_l$, $\lambda_l(i/) = i$, and $\lambda_l(/o) = o$; and $M_r' = M_r\lambda_r$ where $\lambda_r(/) = p_r$, $\lambda_r(i/) = i$, and $\lambda_r(/o) = o$.

Proposition 4.2 *If $M_l' \otimes M_r' \not\models \delta$ and $M_l \succeq (M_\Gamma{\restriction}c_l){\restriction}p_l$ and $M_r \succeq (M_\Gamma{\restriction}c_r){\restriction}p_r$, then $M_\Gamma \not\models \delta$.*

Proof Let $(M_\Gamma{\restriction}c_l){\restriction}p_l$ and $(M_\Gamma{\restriction}c_r){\restriction}p_r$ be given by $M_\Gamma^l = (Q_\Gamma^l, B_\Gamma^l, \Delta_\Gamma^l)$ and $M_\Gamma^r = (Q_\Gamma^r, B_\Gamma^r, \Delta_\Gamma^r)$, respectively. First of all, note that $q \in Q_\Gamma^l$ iff $q \in Q_\Gamma$, $q \in B_\Gamma^l$ iff $q \in B_\Gamma$, and $(q, (l{\restriction}c_l){\restriction}p_l, q') \in \Delta_\Gamma^l$ iff $(q, l, q') \in \Delta_\Gamma$; and analogously for M_Γ^r and M_Γ. Let R_l and R_r be simulations satisfying condition (ii) on refinements from $(M_\Gamma{\restriction}c_l){\restriction}p_l$ to M_l and $(M_\Gamma{\restriction}c_r){\restriction}p_r$ to M_r, respectively. Define a relation $R \subseteq (Q_l \times Q_r) \times Q_\Gamma$ as follows: $((q_l, q_r), q_\Gamma) \in R \iff (q_l, q_\Gamma) \in R_l \wedge (q_r, q_\Gamma) \in R_r$. If we can show that R is a simulation from M_Γ to $M_l' \otimes M_r'$ satisfying condition (ii) on refinements, the claim follows from Lemma 4.1.

For a proof that R is a simulation from M_Γ to $M_l' \otimes M_r'$, let $((q_l, q_r), q_\Gamma) \in R$. Conditions (i–ii) for simulations hold for R, as A is a connector between Σ_{P_l} and Σ_{P_r} and thus all labels in $M_l' \otimes M_r'$ are internal; and all labels in H_Γ are internal labels, as p_l and p_r always exist and are connected. For condition (iii), let $\mathrm{dead}_{H_\Gamma}(q_\Gamma) \neq \emptyset$. Let $q_{\Gamma,0}, \ldots, q_{\Gamma,n}$ be a sequence of states in Q_Γ such that $q_{\Gamma,0} = q_\Gamma$, and for each $0 \leq i < n$ there is a path from $q_{\Gamma,i}$ to $q_{\Gamma,i+1}$ in H_Γ using first only internal labels $/$ and then exactly one synchronisation label s_i, and from $q_{\Gamma,n}$ there is a path to a state d_Γ with no successors in H_Γ such that this path uses only internal labels $/$. We inductively construct $q_{l,i} \in Q_l$ and $q_{r,i} \in Q_r$ such that $((q_{l,i}, q_{r,i}), q_{\Gamma,i}) \in R$ for all $0 \leq i \leq n$ and such that there is a path from $(q_{l,i}, q_{r,i})$ to $(q_{l,i+1}, q_{r,i+1})$ in $M_l' \otimes M_r'$ for all $0 \leq i < n$. If $n = 0$, we can choose $q_{l,0} = q_l$ and $q_{r,0} = q_r$. Let $n > 0$ and let $q_{l,i} \in Q_l$ and $q_{r,i} \in Q_r$ be constructed for $0 \leq i < n$. Then $\mathrm{enabled}_{M_\Gamma^l}(q_{\Gamma,n-1}) \neq \emptyset$ and $\mathrm{enabled}_{M_\Gamma^r}(q_{\Gamma,n-1}) \neq \emptyset$, as s_{n-1} is a synchronisation label; say, $m/ \in \mathrm{enabled}_{M_\Gamma^l}(q_{\Gamma,n-1})$ and $/m \in \mathrm{enabled}_{M_\Gamma^r}(q_{\Gamma,n-1})$. By conditions (i–ii) on simulations, there are $q_{l,n} \in \mathrm{reach}_{M_l}(q_{l,n-1}, m/)$ with $(q_{l,n}, q_{\Gamma,n}) \in R_l$ and $q_{r,n} \in \mathrm{reach}_{M_r}(q_{r,n-1}, /m)$ with $(q_{r,n}, q_{\Gamma,n}) \in R_r$, hence $((q_{l,n}, q_{r,n}), q_{\Gamma,n}) \in R$; and by the construction of $M_l' \otimes M_r'$ there is a path from $(q_{l,n-1}, q_{r,n-1})$ to $(q_{l,n}, q_{r,n})$ in $M_l' \otimes M_r'$. Using such a sequence of states $(q_{l,i}, q_{r,i})_{0 \leq i \leq n}$ in $M_l' \otimes M_r'$ we have that $\mathrm{dead}_{M_l}(q_{l,n}) \neq \emptyset$ and $\mathrm{dead}_{M_r}(q_{r,n}) \neq \emptyset$ and thus $\mathrm{dead}_{M_l' \otimes M_r'}(q_{l,n}, q_{r,n}) \neq \emptyset$ and hence also $\mathrm{dead}_{M_l' \otimes M_r'}(q_{l,0}, q_{r,0}) \neq \emptyset$, as there is a path from $(q_{l,0}, q_{r,0})$ to $(q_{l,n}, q_{r,n})$ in $M_l' \otimes M_r'$ using only internal labels.

Finally, for $q_\Gamma \in B_\Gamma$ there are $q_l \in B_l$ and $q_r \in B_r$ with $(q_l, q_\Gamma) \in R_l$ and $(q_r, q_\Gamma) \in R_r$ by condition (ii) on refinements. Hence, $((q_l, q_r), q_\Gamma) \in R$ and $(q_l, q_r) \in B_{M_l' \otimes M_r'}$. Thus, $M_l' \otimes M_r' \succeq M_\Gamma$. As, by Lemma 4.1, refinements preserve deadlock-freedom,

M_Γ is deadlock-free. □

This result also means that if the parallel composition of the port protocol state machines does not produce a deadlock then the only situation where the assembly can deadlock is by violating the refinement relations. That is, one of the component fails to react to an incoming message or fails to produce an output required by the other component. This could, for example, be the case if the component waited on a message of a different port.

The I/O-transition system of the component $BuildComp(M_\Gamma)$ has the same states and transitions as the I/O-transition system for the assembly M_Γ. Thus Prop. 4.2 leads immediately to:

Corollary 4.3 *Given the assumptions for Prop. 4.2. Then* $BuildComp(M_\Gamma) \not\models \delta$ *for the component* $BuildComp(M_\Gamma)$ *built from the assembly* M_Γ.

4.3 Model Checking

Deadlock-freedom of the parallel composition of two finite port protocols can be checked using the UML state machine model checker HUGO [10]. Which additional verification and abstraction techniques to apply to check infinite port protocols is beyond the scope of this paper. For example, the semantics of the UML state machine in Fig. 3 is the I/O-transition system in Fig. 5, which has infinitely many transitions from Idle to Verifying given by all possible arguments to the operation verifyPin. To apply HUGO to such a system, we may use as an appropriate abstraction the finite state machine in Fig. 9 where all operations are parameter-free. It is easy to see that since the transitions in the original I/O-transition system do not depend on arguments, deadlock-freedom of the abstracted, finite system implies deadlock-freedom of the original system.

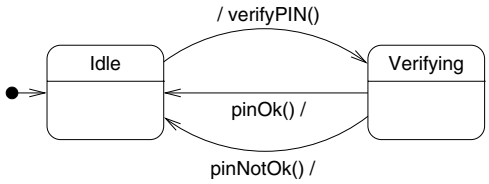


Figure 9. An abstraction of the protocol state machine of the port BC from Fig. 3 used for model checking.

5 Related Work

Architectural programming languages, and thus JAVA/A, transfer long standing ideas from architecture description languages into programming. ADLs [11] support the modelling and, partially, the analysis of component-based systems; furthermore, some ADLs also provide code generation facilities (see, e.g. [15]). Architectural programming languages, however, aim directly at the implementation level and the representation of a system architecture in the code preventing architectural erosion in the implementation and maintenance phases. Therefore, APLs can be seen as a complement to ADLs.

More recently, ArchJava [1] has been proposed as a programming language integrating architectural concepts into Java, similar to JAVA/A. However, ArchJava lacks port protocols or other abstract means to specify component behaviour. Thus, reasoning about communication integrity is limited to analysis on the interface level. Furthermore, ArchJava does not provide a formal semantics.

A different approach to the implementation of software architectures is followed by the component models SOFA [16] and Fractal [3] and the work by Pavel et al. [13]: The structural definition of components, their ports, and assemblies is separated from the behaviour of these components and assemblies. The implementation of component-based software systems is based on frameworks and development guidelines; SOFA also offers a code template generation tool. However, neither approach integrates architectural concepts into a programming language.

There is a variety of technologies to support the implementation of components. Unlike APLs, ADLs, or component models, these technologies—among them Enterprise JavaBeans, COM+, CORBA [17]—leave the structural aspects of software architectures mostly implicit. Moreover, they do not properly support the concepts of (required) interfaces, connectors, and assemblies.

6 Conclusions

We have presented the principles of the architectural programming language JAVA/A which supports component-based implementations of large-scale software systems. The formal foundation of the semantics of JAVA/A given by an abstract component model which uses algebras to denote states and I/O-transition systems to model the behaviour of components. In addition to the JAVA/A implementation [7] a prototype [8] serving as a test bed for our semantic model has been implemented in VisualWorks Smalltalk [18].

Our study provides the basis for challenging, future tasks concerning the consolidation of our component model, the development of a comprehensive specification framework for components and the investigation of correctness notions for component realisations and refinements. For the consolidation of the component model we are particularly interested to integrate complex (n-ary) connectors and to model global reconfigurations and shared objects. Concerning component specifications our approach actually supports the definition of port protocols which focus on the external (black box) views of components. While black-box views are important for the user of a component, glass-box views are important for the implementor of a component. An important issue is to extend our approach to the specification of glass-box views describing the internal behaviour of a component. On this basis we can then define a correctness notion for JAVA/A programs such that a JAVA/A program is a correct realisation of a component specification if it satisfies the required internal behaviour.

[7] http://www.pst.ifi.lmu.de/projekte/javaa
[8] http://www.pst.ifi.lmu.de/~baumeist/components

Acknowledgements

We would like to thank Jeff Kramer and Jeff Magee for valuable hints and discussions.

References

[1] ArchJava. http://www.archjava.org(12/12/05).

[2] Luca de Alfaro and Thomas A. Henzinger. *Interface Automata.* In *Proc. 9th Ann. Symp. Foundations of Software Engineering (FSE'01)*, pages 109–120, Wien, 2001. ACM Press.

[3] Fractal. http://fractal.objectweb.org(12/12/05).

[4] Harald Ganzinger. *Programs as Transformations of Algebraic Theories (Extended Abstract). Informatik Fachberichte*, 50:22–41, 1981.

[5] David Garlan and Mary Shaw. "Software Architecture, Perspectives of an Emerging Discipline". Prentice Hall, New York–&c., 1996.

[6] Martin Große-Rhode. "Semantic Integration of Heterogeneous Software Specifications". Springer-Berlin, 2003.

[7] Florian Hacklinger. JAVA/A – *Taking Components into Java.* In *Proc. 13th ISCA Int. Conf. Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, pages 163–169. ISCA, Cary, NC, 2004.

[8] HUGO. http://www.pst.ifi.lmu.de/projekte/hugo(12/12/05).

[9] IEEE. "IEEE Recommended Practice for Architectural Description of Software-intensive Systems". Standard 1471-2000, IEEE, 2000.

[10] Alexander Knapp, Stephan Merz, and Christopher Rauh. *Model Checking Timed UML State Machines and Collaborations.* In Werner Damm and Ernst Rüdiger Olderog, editors, *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *Lect. Notes Comp. Sci.*, pages 395–416. Springer, Berlin, 2002.

[11] Nenad Medvidovic and Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[12] Object Management Group. *Unified Modeling Language: Superstructure, Version 2.0.* Technical report, OMG, 2004.

[13] Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer. *A Java Implementation of a Component Model with Explicit Symbolic Protocols.* In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Proc. 4th Int. Wsh. Software Composition (SC'05). Rev. Sel. Papers*, volume 3628 of *Lect. Notes Comp. Sci.*, pages 115–124, 2005.

[14] Dewayne E. Perry and Alexander L. Wolf. *Foundations for the Study of Software Architecture. ACM SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[15] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. *Abstractions for Software Architecture and Tools to Support Them. IEEE Trans. Software Eng.*, 21(4):314–335, 1995.

[16] SOFA. http://sofa.objectweb.org(12/12/05).

[17] Clemens Szyperski. "Component Software". Addison-Wesley, Harlow–&c., 2nd edition, 2002.

[18] VisualWorks Smalltalk. http://www.cincomsmalltalk.com(12/12/05).

[19] Martin Wirsing. *Algebraic Specification.* In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 675–788. Elsevier, Amsterdam, 1990.

[20] Daniel M. Yellin and Robert E. Strom. *Protocol Specifications and Component Adaptors. ACM Trans. Prog. Lang. Sys.*, 19(2):292–333, 1997.