

Architectural Interaction Diagrams: AIDs for System Modeling*

Arnab Ray Rance Cleaveland
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY, 11794-4400 USA
{arnabray,rance}@cs.sunysb.edu

Abstract

This paper develops a modeling paradigm called Architectural Interaction Diagrams, or AIDs, for the high-level design of systems containing concurrent, interacting components. The novelty of AIDs is that they introduce interaction mechanisms, or buses, as first-class entities into the modeling vocabulary. Users then have the capability, in their modeling, of using buses whose behavior captures interaction at a higher level of abstraction than that afforded by modeling notations such as Message Sequence Charts or process algebra, which typically provide only one fixed interaction mechanism. This paper defines AIDs formally by giving them an operational semantics that describes how buses combine subsystem transitions into system-level transitions. This semantics enables AIDs to be simulated; to incorporate subsystems given in different modeling notations into a single system model; and to use testing, debugging and model checking early in the system design cycle in order to catch design errors before they are implemented.

1. Introduction

Pre-implementation modeling of software systems has been accepted as good software engineering practice for some time. The utility of such *design-time* artifacts is that they allow users to model systems at a high level, without worrying about implementation details, in order to study high-level structure and behavior and isolate design bugs early in the development cycle. While not nearly as widespread as in other areas of engineering, the use of modeling tools for software systems is growing rapidly, as notations such as Statecharts [6] and Message Sequence Charts (MSCs) [1], and commercial languages such as UML [3]

and Simulink®/ Stateflow®¹ [7], gain adherents.

Despite the many benefits of modeling, however, one may identify a common shortcoming *vis à vis* modeling notations for concurrent and distributed systems: they force users to adopt low-level communication primitives too early in the design cycle. For example, in Statecharts, synchronous broadcast is the only communication mechanism, while in MSCs only asynchronous point-to-point message-passing is allowed. Given the diversity of high-level communications primitives available to implementors (multi-casting, shared memory, CANbus, HTTP, etc.), the paucity of interaction mechanisms in these modeling notations severely limits their usefulness in modeling systems containing interacting components. In particular, the problem that arises is that the interaction discipline of the underlying modeling language becomes tightly bound to the system modeling, which becomes guided more by what the modeling language permits rather than the *actual* interprocess communication mechanism the system designer might have access to during implementation.

The goal of this paper is to devise a formal framework in which high-level interaction mechanisms can be defined and used to support the mechanical analysis of system behavior. Our theory, which we call *Architectural Interaction Diagrams* (AIDs), supports a highly parameterizable notion of interprocess communication, freeing designers from artificial constraints on how his/her design components may interact with one another. The definition of the notation also allows submodels given in different notations to interoperate. In contrast to existing modeling notations, the framework is extensible in that it allows the user to implement his/her own interprocess communication; new interprocess primitives can be added and made a part of the language itself.

Our approach also supports the decoupling of a system's interactions from the internal workings of its subsystems. This decoupling of interaction from behavior has

*Research supported by Army Research Office grants DAAD190110003 and DAAD190110019, and National Science Foundation grants CCR-9988489 and CCR-0098037.

¹Simulink and Stateflow are registered trademarks of The MathWorks, Inc.

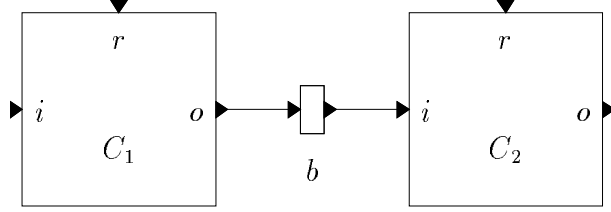


Figure 1. A sample architecture interaction diagram.

great significance in terms of platform-independent modeling of software systems. One of the important issues that arises when porting software from one platform to another is the different inter-process communication primitives provided by the new platform. In conventional modeling, the software components would have to be re-examined and re-adjusted to take into account the new system of inter process communication. In AIDs, all the designer has to do is to use system-defined buses or his/her own definitions without having to adjust the different software components themselves. Of course, it would still be incumbent on the designer to verify that changing the buses still yields correct over-all behavior.

The rest of the paper is arranged as follows. The next section gives an intuitive introduction to AIDs and discusses related work. We then give a formalization of AIDs in Section 3, including a precise definition of the structure of AIDs and how they execute. Section 4 discusses some example interaction mechanisms definable in AIDs, while the final section presents future work and our conclusions.

2. An Introduction to AIDs

The goal of AIDs is to give precise mathematical meaning to diagrams such as the one in Fig. 1. The diagram depicts a model containing two component models, C_1 and C_2 . Each component is embedded inside an interface containing input (i , r) and output (o) “ports” that the components may use to interact with their environments. These ports may be left unlinked, indicating an “open connection”, or they may be linked to *interaction mechanisms*, or *buses* (b). Buses contain their own ports and are responsible for combining submodel transitions involving the bus’s ports into system-level transitions. A main goal of this paper is to give a highly flexible framework for defining buses that permits many different interaction disciplines to be captured uniformly.

Fig. 2 illustrates another important aspect of AIDs, namely, that models can be nested hierarchically, with component ports “exported” to enclosing interfaces. This figure also contains a bus, q , with a different graphical representation than b . The intention is that q represents a FIFO reliable channel, while b represents synchronous hand-shaking.

Thus, the model in this diagram employs two different communication mechanisms.

Related Work. AIDs builds on the Graphical Calculus of Communicating Systems (GCCS) presented in [4]. Like AIDs, GCCS supports the hierarchical construction of models of communicating subsystems, and the language is also given a formal operational semantics in which subsystem transitions are combined into system transitions. However, GCCS includes only a single interaction mechanism, binary handshaking, and thus suffers from the same drawbacks for modeling interaction that the languages mentioned in the introduction do.

Our approach is also akin to that adopted in [13], where *infra-models* are proposed as a mechanism for supporting interoperability among different modeling notations. Infra-models share a syntactic similarity to Petri Nets: such models are hypergraphs consisting of nodes (= places, in Petri-Net terminology) and hyperedges (= transitions). Infra-models augment standard Petri Nets by an ability to define firing rules that are different from traditional Petri Net firing rules; these are used to capture different notions of synchrony. However, the work lacks a precise mathematical account of how infra-models interact with component models.

AIDs also share a similarity to the *architectural styles* work presented in [5, 16]. There, the authors define architectural styles to be reference models for system architectures, and they observe that architectural styles exhibit four kinds of properties.

1. They provide a vocabulary of design elements, which are classified in their terminology as component and connector types.
2. They define a set of configuration rules or topological constraints. Examples of topological constraints are if the processes engage in single or multi-party communication.
3. They define a semantic interpretation for compositions of components that give such compositions a well defined meaning.

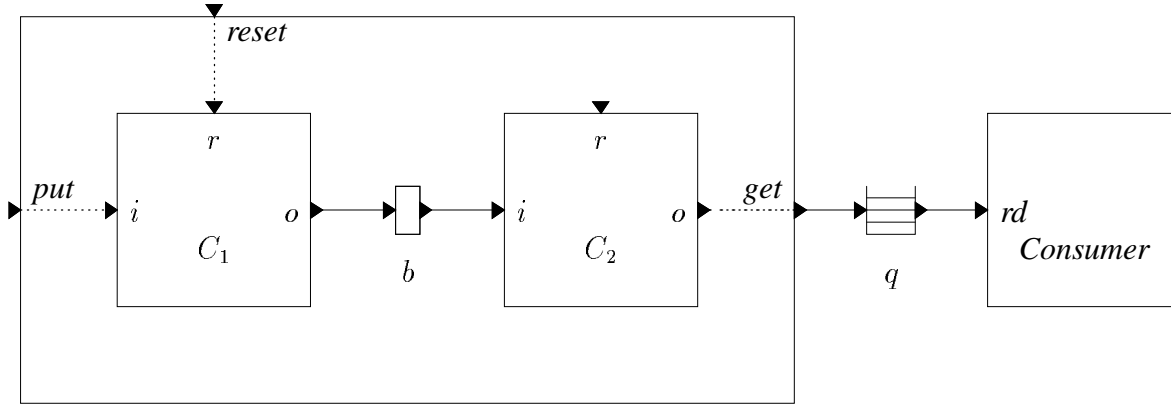


Figure 2. A nested AID.

4. They define analyses that can be performed on systems built in that style.

AIDs can be readily thought of as a tool for creating architectural styles according to the criteria above. Component types for AIDs consist of the processes that are plugged in while the connector types denote the different kind of buses that are supported by the framework. However, AIDs supports a much wider range of interaction disciplines than traditional architectural description languages.

Like AIDs, WRIGHT [15, 16] also treats interprocess communication structures (called connectors) as first class entities of the modeling vocabulary. Connectors in WRIGHT are semantically same as the components; to define a new type of connector, one “implements” it as a process, using CSP’s native handshake coordination mechanism. (The reason for this is that WRIGHT is built on top of CSP.) The disadvantage of this approach in relation to ours is that these implementations can be subtle (think of implementing multi-casting using handshaking) and error-prone. In contrast, buses in AIDs are given a much more declarative and concise semantics. The other disadvantage involves state explosion. In general, WRIGHT connectors will contain significant internal computation, since they must encode the intended interaction semantics in terms of CSP’s. A single interaction among components involving such a connector may therefore give rise to a number of transitions, with the possibilities of interleavings with other ongoing interactions leading to significant state-space growth. In contrast, our semantics guarantees that any interaction involving a bus generates exactly one system-level transition.

The aims of the Ptolemy project [10] are very similar to those for AIDs. In a Ptolemy model, a system model contains a topology of subsystems and interaction mechanisms, together with a definition of how the interaction mechanisms cause submodels to interact. Thus all interaction devices at the same “level” of a model have the same se-

mantics. The advantage of AIDs is that different buses may be freely intermixed at the same level in a system model; in Ptolemy, multiple levels of submodeling would have to be introduced to achieve this effect. In addition, the AIDs semantic framework for buses gives a uniform paradigm for defining interaction behavior; the Ptolemy approach, on the other hand, is rather more *ad hoc*, making it difficult for users to add to the interaction mechanisms supported.

Coordination languages have generally been applied in the domain of programming languages in the coding phase of the software life cycle. Examples of such coordination languages in the programming language domain include Linda [12], Javaspaces [8] and TSpaces [14]. AIDs, as mentioned before, are a design notation, although they share some of the same motivations as coordination languages, specifically, the capability to integrate subsystems given in different languages into a single system.

Process-algebra-based tools like the CWB-NC [17] provide support for specifying architectural designs in a variety of input languages. Unlike traditional architecture description languages, process algebras have a precise semantics, and indeed have been used to define the semantics of several ADLs [2, 11, 16]. They also have a capacity for hierarchical modeling. However, process algebras, like GCCS, typically only include a single primitive process-interaction mechanism.

3. Formalizing AIDs

This section is given over to a mathematically rigorous definition of AIDs and how they execute. The goal of this effort is, first, to make precise what the different elements of an AID are, and second, to develop a foundation for mechanically analyzing their behavior. In the rest of this paper we fix the following.

- \mathbb{W} is an infinite set of *write ports*.

- \mathbb{R} is an infinite set of *read ports*, with $\mathbb{R} \cap \mathbb{W} = \emptyset$.
- \mathbb{V} is a nonempty set of *values*.

Intuitively, \mathbb{W} and \mathbb{R} contain all the possible port names that may be used to define a given system, while \mathbb{V} contains all the values that may be used. In this paper our focus is on interaction rather than data manipulation, so we do not impose any additional structure on \mathbb{V} . We also use the following definitions in what follows.

- $\mathbb{O} = \{w!v \mid w \in \mathbb{W}, v \in \mathbb{V}\}$ is the set of *output actions*.
- $\mathbb{I} = \{r? \mid r \in \mathbb{R}\}$ is the set of *input actions*.

The sets \mathbb{O} and \mathbb{I} represent interactions that a system may engage in with its environment: outputting, and inputting.

3.1. I/O Labeled Transition Systems

The basic components of the AID theory are *I/O labeled transition systems* (IOLTSs). These are defined as follows.

Definition 3.1 An I/O labeled transition system (IOLTS) is a tuple $\langle Q, T, q_0 \rangle$, where Q is a set of states, $q_0 \in Q$ is the start state, and $T = T_{\text{write}} \cup T_{\text{read}} \cup T_{\text{internal}}$ is the transition relation such that:

1. $T_{\text{write}} \subseteq Q \times \mathbb{O} \times Q$,
2. $T_{\text{read}} \subseteq Q \times \mathbb{I} \times (\mathbb{V} \rightarrow Q)$, and
3. $T_{\text{internal}} \subseteq Q \times Q$.

An IOLTS encodes the operational behavior of a system, with Q being the set of system states and q_0 the initial state. State transitions may take one of three forms.

- An *output transition* $\langle q, w!v, q' \rangle \in T_{\text{write}}$ indicates a state change from q to q' when value v is written out to the environment on write port w .
- In *input transition* $\langle q, r?, f \rangle \in T_{\text{read}}$, f is a function mapping values to states. This transition indicates a state change from q to $f(v)$ if the system's environment supplies value v on read port r .
- An *internal transition* $\langle q, q' \rangle \in T_{\text{internal}}$ represents an execution step that the system can engage in without any interaction with its environment.

In what follows, if $P = \langle Q, T, q_0 \rangle$ is an IOLTS then we write:

- $q \xrightarrow{w!v}_P q'$ if $\langle q, w!v, q' \rangle \in T$;
- $q \xrightarrow{r?}_P f$ if $\langle q, r?, f \rangle \in T$; and

- $q \xrightarrow{\tau}_P q'$ if $\langle q, q' \rangle \in T$.

If $t \in T$ then we use $\text{src}(t), \text{tgt}(t) \in Q \cup (\mathbb{V} \rightarrow Q)$ to refer to the source and target of t : $\text{src}(t) = q$ and $\text{tgt}(t) = q'$ if $t = \langle q, w!v, q' \rangle$ or $t = \langle q, q' \rangle$, and $\text{src}(t) = q$ and $\text{tgt}(t) = f$ if $t = \langle q, r?, f \rangle$. We also define the following.

- $Q(P)$ refers to the state set of P , while $q_0(P)$ is the initial state of P .
- $T_{\text{write}}(P), T_{\text{read}}(P)$ and $T_{\text{internal}}(P)$ refer to the write, read and internal transition sets of P (so $T = T_{\text{write}}(P) \cup T_{\text{read}}(P) \cup T_{\text{internal}}(P)$).
- The *write interface*, $WI(P) \subseteq \mathbb{W}$, of P is the set of write ports that P might use as it executes:

$$WI(P) = \{w \in \mathbb{W} \mid \exists q, q' \in Q, v \in \mathbb{V}. q \xrightarrow{w!v}_P q'\}.$$

- The *read interface*, $RI(P) \subseteq \mathbb{R}$, of P is the set of read ports that P might use as it executes:

$$RI(P) = \{r \in \mathbb{R} \mid \exists q \in Q, f \in \mathbb{V} \rightarrow Q. q \xrightarrow{r?}_P f\}.$$

3.2. Defining AIDs

We now define the structure of AIDs. An AID may take one of two forms.

1. It may be an IOLTS.
2. It may be a *network* containing other AIDs embedded in interfaces and connected together into a communication topology, as depicted in Figs. 1 and 2.

The theory imposes no restrictions on how an IOLTS AID is described concretely: it could be a Statechart, or a term in process algebra, or a program. The only requirement is that the model have identifiable read and write transitions.

The remainder of this section is devoted to defining the structure of networks. From the figures, one may identify the following components of a network AID.

- AIDs describing subsystems.
- Interfaces, containing read and write ports, surrounding subsystems.
- Connections between ports in a subsystem and ports in an interface (cf. the dotted lines in Fig. 2).
- Buses.
- Links from interface ports to buses.

We first give mathematical accounts of interfaces and buses and then use them to define networks formally.

Interfaces. An interface consists of a set of read ports and a set of write ports. Mathematically, an interface I is a pair $\langle W, R \rangle$, where $W \subseteq \mathbb{W}$ and $R \subseteq \mathbb{R}$ are the *write and read interfaces*, respectively, of I . We write $W(I)$ for the write interface of I and $R(I)$ for the read interface of I .

Buses. In AIDs buses handle interactions between subsystems. As such, they have two responsibilities: the transfer of data between senders and receivers, and the synchronization of sender/receiver transitions, depending on the semantics of the interaction mechanism. For example, consider a synchronous binary handshaking interaction mechanism. Not only must a bus implementing this mechanism deliver a data value from a sender to a receiver, but it must also ensure that senders and receivers block until a communication partner is ready to execute. In the case of bounded-buffer non-lossy communication, on the other hand, senders should be blocked when the buffer is full, while receivers should be blocked when the buffer is empty. In shared memory neither senders (“writers”) nor receivers (“readers”) ever block. Providing a common framework for explicating these subtleties is a central goal of the AIDs theory.

Our treatment of buses is inspired by the work on *action transducers* of [9]. In particular, we wish to view buses as “transducers” that combine transitions of subsystems connected to the bus into system-level transitions, according to the synchronization discipline the bus is intended to capture. These considerations underly the following definition.

Definition 3.2 A bus is a tuple of form $\langle I, B, T, b_0 \rangle$, where I is an interface, B is a set of states, $T \subseteq B \times 2^{(W(I) \times \mathbb{V})} \times 2^{R(I)} \times 2^{W(I)} \times 2^{(R(I) \times \mathbb{V})} \times B$ is the transition relation, and $b_0 \in B$ is the start state. Every transition $\langle q, WV, R, W, RV, q' \rangle \in T$ is also required to satisfy the following:

$$\begin{aligned} R &\supseteq \{r \mid \exists v \in \mathbb{V}. \langle r, v \rangle \in RV\} \\ W &\subseteq \{w \mid \exists v \in \mathbb{V}. \langle w, v \rangle \in WV\} \end{aligned}$$

If $M = \langle I, B, T, b_0 \rangle$ is a bus and $\langle b, WV, R, RV, b' \rangle \in T$ is a transition, then we write

$$b \xrightarrow[WV R]{W RV} b'$$

We also write $W(M) = W(I)$ and $R(M) = R(I)$ for the write and read interfaces, respectively, of M , and $B(M)$, $T(M)$ and $b_0(M)$ for the state set, transition relation, and initial state of M , respectively.

Intuitively, a bus contains a read/write interface, a set of states reflecting the internal status of the bus, a transition relation, and an initial state. Buses are similar to IOLTSSs, but the transition relation is significantly different and requires more comment. A bus transition of form

$$b \xrightarrow[WV R]{W RV} b'$$

is intended to be read as: “if the bus is in state b , and subsystems connected to the bus enable write transitions as indicated in WV and read transitions as enabled in R , then the bus fires read transitions as indicated in RV and write transitions as indicated in W and goes to state b' .” This firing of selected read and write transitions in systems connected to the bus is also done atomically: thus one bus transition may “consume” several transitions from the components connected to it. Also, “writing” to a bus is interpreted with respect to components connected to a bus: so write ports on a subsystem are connected to write ports on a bus, and similarly for read ports.

AIDs. We now have the machinery needed to define AIDs. The definition below develops two notions simultaneously: AID, and the principal read/write interface of an AID.

Definition 3.3 An architecture interaction diagram (AID) is either:

1. an IOLTS P , with principal write interface $PWI(P) = WI(P)$ and principal read interface $PR I(P) = RI(P)$; or

2. a network $N = \langle \bar{C}, \bar{M}, L \rangle$, where:

(a) $\bar{C} = \langle C_1, \dots, C_n \rangle$ is a tuple of components, where each $C_i = \langle S_i, I_i, G_i \rangle$ consists of an AID S_i , an interface I_i , and a gate definition G_i (gate definitions are defined below);

(b) $\bar{M} = \langle M_1, \dots, M_k \rangle$ is a tuple of buses; and

(c) $L \subseteq (\{1, \dots, n\} \times (\mathbb{W} \cup \mathbb{R})) \times (\{1, \dots, k\} \times (\mathbb{W} \cup \mathbb{R}))$, the link set, satisfies: if $\langle \langle i, p_1 \rangle, \langle j, p_2 \rangle \rangle \in L$ then

i. $p_1 \in \mathbb{W}$ if and only if $p_2 \in \mathbb{W}$;

ii. $p_1 \in W(I_i) \cup R(I_i)$;

iii. $p_2 \in W(M_j) \cup R(M_j)$;

iv. $\langle \langle \ell, p'_1 \rangle, \langle j, p_2 \rangle \rangle \in L$ iff $\ell = i$ and $p'_1 = p_1$.

v. $\langle \langle i, p_1 \rangle, \langle \ell, p'_2 \rangle \rangle \in L$ iff $\ell = j$ and $p'_2 = p_2$.

$PWI(N) = \{\langle i, w \rangle \mid w \in W(I_i) \wedge \forall j, w'. \langle \langle i, w \rangle, \langle j, w' \rangle \rangle \notin L\}$ is the principal write interface of N , while $PR I(N) = \{\langle i, r \rangle \mid r \in R(I_i) \wedge \forall j, r'. \langle \langle i, r \rangle, \langle j, r' \rangle \rangle \notin L\}$ is the principal read interface. Also, $|N|_C = n$ is the number of components in N , while $|N|_M$ is the number of buses.

Intuitively, a network contains a list of components, each containing a subsystem description, an interface, and a *gate* (see below) that defines how the ports of the subsystem are connected to the interface. It also contains a list of buses and a link set connecting component ports to bus ports so

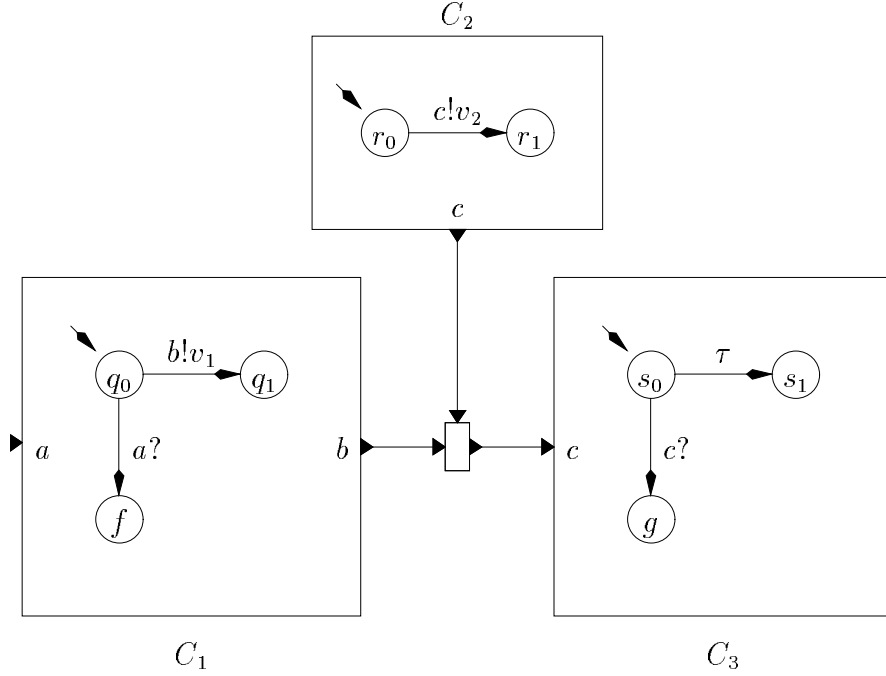


Figure 3. A network AID; subsystems are IOLTSSs.

that: write ports are connected to write ports, and read ports to read ports, and each port (bus or component) has at most one link to it. The principal write and read interfaces of N contain the component number / port pairs describing ports on given interfaces that are not connected to a bus by a link.

We now define gates as follows.

Definition 3.4 Let S be an AID and I an interface. Then a gate $G \in (PWI(S) \cup PRI(S)) \rightarrow (W(I) \cup R(I))$ is a one-to-one partial function satisfying:

1. If $p \in PWI(S)$ and $G(p)$ defined, then $G(p) \in W(I)$.
2. If $p \in PRI(S)$ and $G(p)$ defined, then $G(p) \in R(I)$.

This definition defines a gate to be a connection between write ports in a system and write ports in an (enclosing) interface, and similarly for read ports. The fact that G is a one-to-one partial function guarantees that every port in S is connected to at most one port in I , and vice versa. Note that if S is an IOLTS, then $PWI(S)$ and $PRI(S)$ contain ports, while if S is a network then these sets contain pairs comprising an interface index and a port. Also note that since G is one-to-one, its inverse G^{-1} is a partial function.

Finally, we use the following notation in what follows. If N is a network then we write $C(N, i)$ for the i^{th} component of N and $M(N, i)$ for the i^{th} bus. If $C(N, i) = \langle S_i, I_i, G_i \rangle$ then we use $S(N, i)$, $I(N, i)$ and $G(N, i)$ to refer to S_i , I_i and G_i , respectively.

3.3. Semantics of AIDs

A chief aim of this paper is to equip AIDs with an operational semantics describing the execution steps that individual AIDs may engage in. As an example of what this semantics should do, consider the network AID N in Fig. 3, which consists of three components, each subsystem of which is an IOLTS. Also assume that the bus implements binary synchronous handshaking. If we represent the current state of the system as a vector of system / bus states, and the bus only contains one state, b_0 , then semantics should allow us to infer execution steps like the following.

$$\langle q_0, r_0, s_0, b_0 \rangle \xrightarrow{a?}_N \lambda v. \langle f(v), r_0, s_0, b_0 \rangle \quad (1)$$

$$\langle q_0, r_0, s_0, b_0 \rangle \xrightarrow{\tau}_N \langle q_0, r_0, s_1, b_0 \rangle \quad (2)$$

$$\langle q_0, r_0, s_0, b_0 \rangle \xrightarrow{\tau}_N \langle q_1, r_0, g(v_1), b_0 \rangle \quad (3)$$

$$\langle q_0, r_0, s_0, b_0 \rangle \xrightarrow{\tau}_N \langle q_0, r_1, g(v_2), b_0 \rangle \quad (4)$$

In Step 1, component C_1 awaits an input from its (unconnected) port a while the other components idle (the λ expression is the standard *anonymous function* notation from the λ -calculus; it is explained later). Similarly, in Step 2, C_3 executes its internal computation step while the other components idle. The last two steps show synchronizations over the bus between C_1 and C_3 (Step 3), and between C_2 and C_3 (Step 4).

Mathematically, we define the semantics in the Structural Operational Semantic (SOS) style, by showing how

a single AID may be viewed as an IOLTS whose transition relation reflects the execution steps of the AID. Also in line with SOS, the definition of the transition relation of an AID is given inductively using inference rules that explain how transitions of subsystems are combined to form transitions of systems. More precisely, given an AID N we wish to associate with N an IOLTS $\langle Q_N, T_N, q_N \rangle$. If $N = \langle Q, T, q_0 \rangle$ is itself an IOLTS, the association is obvious: take $Q_N = Q$, $T_N = T$ and $q_N = q_0$.

Now suppose that $N = \langle \bar{C}, \bar{M}, L \rangle$. What should Q_N, T_N and q_N be? In the case of Q_N , each system state should record current state information for each component and bus, and the initial state should contain the initial states of each component and bus. This leads to the following.

Definition 3.5 *Let $N = \langle \bar{C}, \bar{M}, L \rangle$ be a network AID, and let $n = |N|_C$ and $k = |N|_M$. Then:*

1. $Q_N = C_N \times M_N$, where:

$$\begin{aligned} C_N &= \{ \langle q_1, \dots, q_n \rangle \mid q_i \in Q(S_i) \} \\ M_N &= \{ \langle b_1, \dots, b_k \rangle \mid b_i \in B(M_i) \} \end{aligned}$$

2. $q_N = C_N^0 \times M_N^0$, where $C_N^0 = \langle q_0(S_1), \dots, q_0(S_n) \rangle$ and $M_N^0 = \langle b_0(M_1), \dots, b_0(M_k) \rangle$.

Thus, the states for N 's IOLTS consists of a state vector for N 's components and another state vector for N 's buses, with the start state for N containing the start states for each component and bus. We often represent these states as pairs $\langle \bar{s}, \bar{b} \rangle$, where \bar{s} and \bar{b} are the subsystem- and bus-state vectors, respectively. We also use the following notation on vectors. Let $\bar{v} = \langle v_1, \dots, v_n \rangle$ be a vector. Then:

- $\bar{v}[i] = v_i$.
- $\bar{v}[i := u]$ is the vector obtained by modifying the i^{th} component of \bar{v} to be u :

$$\bar{v}[i := u] = \langle v_1, \dots, v_{i-1}, u, v_{i+1}, \dots, v_n \rangle.$$

It remains now to define T_N . This is done using SOS rules, each of which has the following general form.

$$\frac{\text{Premises}}{\text{Conclusion}}$$

Intuitively, a rule states that when the premises are true, the conclusion holds. In our case, a conclusion will state the existence of an element of the transition relation, T_N , while the premises will refer to transitions of subsystems and buses as well as conditions about the structure of N .

In preparation for defining T_N , we first give rules showing how transitions of subsystems may be “lifted” to transitions of components. Recall that a component has form $C = \langle S, I, G \rangle$, where S is an AID, I is an interface, and G

is a gate function. The idea behind these rules is to convert transitions of S , which involve S 's ports, into “equivalent” transitions that involve ports on I . In what follows we fix $C = \langle S, I, G \rangle$.

The first two SOS rules define write transitions for C . The first rule handles the case when S is an IOLTS

$$\frac{w \in W(I), G^{-1}(w) = w', q \xrightarrow{w'!v}_S q'}{q \xrightarrow{w!v}_C q'}$$

This rule says that a component C can engage in an output transition labeled $w!v$ if: w is part of the write interface of I ; and the IOLTS S can output v on a port w' that is connected to w (captured by the predicate $G^{-1}(w) = w'$).

The next rule handles the case in which the write port in question is on a subsystem that is a network rather than an IOLTS. To emphasize the fact that we are referring to a network we consider C as $C = \langle N, I, G \rangle$

$$\frac{w \in W(I), G^{-1}(w) = \langle i, w' \rangle, \bar{s}[i] \xrightarrow{w'!v}_{S(N,i)} q'_i}{\langle \bar{s}, \bar{b} \rangle \xrightarrow{w!v}_C \langle \bar{s}[i := q'_i], \bar{b} \rangle}$$

This rule is more complicated than the previous one because the port of S to which w is connected (w') exists on a specific interface within N (the i^{th} one, in the rule), and the rule must ensure that it is the i^{th} subsystem of N that writes on w' . Note that $\bar{s}[i]$ is the state component of the i^{th} subsystem within N .

The next two rules define the read transitions for C . The rules are similar to the ones above in that they define how S 's read transitions may be “lifted” to component-level read transitions. Since the targets of read transitions must be functions, however, the rules use notation, from the λ -calculus, for defining anonymous functions: $\lambda x. e$ is a function that, when applied to a value v , returns the result of evaluating expression e with v substituted for x . The first rule handles the case in which the subsystem contributing the read transition is an IOLTS.

$$\frac{r \in R(I), G^{-1}(r) = r', q \xrightarrow{r'??}_S f}{q \xrightarrow{r??}_C f}$$

This rule says that if read port r has a gate connection to a read port in an IOLTS subsystem, and the IOLTS subsystem can perform a read transition on its read port, then the overall system can perform a read transition involving r .

The second rule handles the situation in which S is a network N .

$$\frac{r \in R(I), G^{-1}(r) = \langle i, r' \rangle, \bar{s}[i] \xrightarrow{r'??}_{S(N,i)} f_i}{\langle \bar{s}, \bar{b} \rangle \xrightarrow{r??}_C \lambda v. \langle \bar{s}[i := f_i(v)], \bar{b} \rangle}$$

The greater complexity of this rule is driven by the same considerations involving the analogous write-transition rule

defined above. Note again that any value input as a result of a read transition is “fed into” the subsystem responsible for engendering the transition.

The last rule handles internal transitions of S .

$$\frac{q \xrightarrow{\tau} S q'}{q \xrightarrow{\tau} C q'}$$

Internal transitions of S become internal transitions of C .

Having defined component transitions, we may now give the SOS rules for a network N . The first defines $(T_{\text{write}})_N$ in terms of component-level write transitions.

$$\frac{\langle i, w \rangle \in PWI(N), s[i] \xrightarrow{w!v}_{C(N,i)} q'_i}{\langle \bar{s}, \bar{b} \rangle \xrightarrow{w!v}_N \langle \bar{s}[i := q'_i], \bar{b} \rangle}$$

This rule states that if a component can engage in a write transition, then N can as well, with all other components remaining idle, provided that the write port involved in the ocmponent transition is not connected to a bus. This last condition is captured formally by the requirement that $\langle i, w \rangle \in PWI(N)$, i.e. that port w on interface I_i is in the principal write interface of N .

The second rule defines $(T_{\text{read}})_N$ similarly.

$$\frac{\langle i, r \rangle \in PRI(N), s[i] \xrightarrow{r?}_{C(N,i)} f'_i}{\langle \bar{s}, \bar{b} \rangle \xrightarrow{r?}_N \lambda v. \langle \bar{s}[i := f'_i(v)], \bar{b} \rangle}$$

Note that the target of the transition is a function that, given a value v , “routes” v to the i^{th} component of N .

The third rules lifts internal component transitions to network-level ones.

$$\frac{s[i] \xrightarrow{\tau}_{C(N,i)} q'_i}{\langle \bar{s}, \bar{b} \rangle \xrightarrow{\tau}_N \langle \bar{s}[i := q'_i], \bar{b} \rangle}$$

The final rule is the most important of the semantics, since it defines how buses combine subsystem transitions into system transitions. The basic idea underlying the rule is this. A bus transition may be thought of as consisting of an “enabling condition” and a “firing condition”. The former requires that certain transitions be enabled on component ports that are connected to different bus ports. The latter then indicates which of the enabled transitions actually fire when the bus transition fires, thus causing state changes in the components as well as the bus.

To capture this intuition formally, we first introduce some auxiliary notions.

Definition 3.6 *Let N be a network.*

1. An annotated component transition of N is a pair $\langle i, t \rangle$ such that $t \in T_{C(N,i)}$ is a transition of the i^{th} component of N . We use AT_N to represent the set of all annotated component transitions of N .

2. A set A of annotated transitions of N is enabled in system state $\langle \bar{s}, \bar{b} \rangle$ if, for every $\langle i, t \rangle \in A$, $\text{src}(t) = \bar{s}[i]$.
3. A set $A \subseteq AT_N$ is independent if for every $\langle i_1, t_1 \rangle, \langle i_2, t_2 \rangle \in A$, if $i_1 = i_2$ then $t_1 = t_2$. In other words, A is independent if every component contributes at most one transition to A .
4. A set $A \subseteq AT_N$ is maximally independent and enabled for system state $\langle \bar{s}, \bar{b} \rangle$ if it is independent and enabled in $\langle \bar{s}, \bar{b} \rangle$ and no proper superset of A satisfies these properties. We write $MIE(N, \langle \bar{s}, \bar{b} \rangle, A)$ when this is the case.
5. Let $A \subseteq AT_N$. Then the forward image, $F(A, N)$ of A along the link set of N is defined as follows.

$$\begin{aligned} F(A, N) &= F_w(A, N) \cup F_r(A, N) \text{ where} \\ F_w(A, N) &= \{ \langle j, w, v \rangle \mid \exists \langle i, t \rangle \in A. t = \langle q, w'!v, q' \rangle \\ &\quad \wedge \langle \langle i, w' \rangle, \langle j, w \rangle \rangle \in L(N) \} \\ F_r(A, N) &= \{ \langle j, r \rangle \mid \exists \langle i, t \rangle \in A. t = \langle q, r'?f \rangle \\ &\quad \wedge \langle \langle i, r' \rangle, \langle j, r \rangle \rangle \in L(N) \} \end{aligned}$$

Intuitively, $F(A, N)$ records the bus ports (and values) used by a set of annotated transitions.

6. Let $t = \langle b, WV, R, RV, W, b' \rangle \in T(M(N, j))$ be a transition of the j^{th} bus in network N . Also let $A \subseteq AT_N$. Then A triggers t if the following hold.
 - (a) $F_w(A, N) \cap \{ \langle j, w, v \rangle \mid w \in W(M(N, j)), v \in \mathbb{V} \} = \{ \langle j, w, v \rangle \mid \langle w, v \rangle \in WV \}$.
 - (b) $F_r(A, N) \cap \{ \langle j, r \rangle \mid r \in R(M(N, j)), v \in \mathbb{V} \} = \{ \langle j, r \rangle \mid r \in R \}$.

Intuitively, A triggers t if the only actions involving $M(N, j)$ that A includes are exactly those encoded in WV and R .

7. Let $t = \langle b, WV, R, RV, W, b' \rangle \in T(M(N, j))$ be a bus transition, and let $A \subseteq AT_N$ be such that A is independent and A triggers t . Also let $\langle \bar{s}, \bar{b} \rangle$ be a state of N . Then $APPLY(A, j, t, \langle \bar{s}, \bar{b} \rangle) = \langle \bar{s}', \bar{b}[j := b'] \rangle$, where \bar{s}' is defined as follows for $1 \leq i \leq |N|_C$.

$$\bar{s}'[i] = \begin{cases} s'_i & \text{if } \exists t, j, v. \langle i, t \rangle \in A, \\ & \langle j, w, v \rangle \in F_w(\{t\}, N), \\ & w \in W \text{ and } s'_i = \text{tgt}(t) \\ f_i(v) & \text{if } \exists t, j, v. \langle i, t \rangle \in A, \\ & \langle j, r \rangle \in F_r(\{t\}, N), \\ & \langle r, v \rangle \in RV \text{ and } f_i = \text{tgt}(t) \\ s_i & \text{otherwise} \end{cases}$$

Intuitively, $APPLY(A, j, t, \langle \bar{s}, \bar{b} \rangle)$ uses bus transition t in bus j , together with the transitions in A that involve ports in the “firing condition” of t , to update the system state. Write transitions in A whose ports are connected to “firing ports” in t execute, causing their components’ sub-states to change. Read transitions in A whose ports are similarly connected also fire, drawing values from RV in order to update their state. Finally, the bus state changes as indicated by the bus transition.

With these definitions, we can now state the final SOS rule for networks as follows.

$$\frac{t = \langle \bar{b}[j], WV, R, RV, W, b' \rangle \in T(B(N, j)), \quad MIE(N, \langle \bar{s}, \bar{b} \rangle, A), \quad A \text{ triggers } t}{\langle \bar{s}, \bar{b} \rangle \xrightarrow{t}_N APPLY(A, j, t, \langle \bar{s}, \bar{b} \rangle)}$$

4. Examples

This section presents several concrete examples of different interaction mechanisms that may be encoded as buses in AIDs. These examples are intended to illustrate the expressiveness of the AIDs theory, on the one hand, and also the relative ease with which new forms of interprocess communication may be defined within the theory.

4.1. Synchronous message passing

The Graphical Calculus of Communicating Systems (GCCS) [4] supports synchronous message passing as its only form of communication. This form of communication is common in process algebras as well, and we show how it may be encoded as a bus.

Buses in GCCS place no limit on how many subsystems are allowed to use them. They require all senders and receivers to block until at least one sender and receiver are enabled; then an exchange of data occurs, with the selected sender and receiver free to continue executing.

A bus $M_S = \langle I, B, T, b_0 \rangle$ encapsulating synchronous binary handshaking may be defined as follows.

- $I = \langle \mathbb{W}, \mathbb{R} \rangle$. This reflects the fact that there is no limit on the number of links to M_S .
- $B = \{b\}$ consists of a single state.
- T is defined below.
- $b_0 = b$.

T contains all transitions of the form

$$b \xrightarrow[WV R]{W RV} b$$

where:

- $WV \neq \emptyset, R \neq \emptyset$
- $\exists \langle w, v \rangle \in WV, r \in R. W = \{w\}, RV = \{\langle r, v \rangle\}$.

In other words, a bus transition is enabled any time there is at least one reader and writer, and the result of firing the transition is to cause exactly one writer and one reader to execute, with the value output by the writer being shifted to the reader. Note that the bus never changes state; the only role of M_S 's transitions is to synchronize the transitions of users of the bus.

One may verify that the SOS rules for AIDs ensure that, if several processes are connected to bus M_S , then all interaction consists of binary handshaking.

Note that connection limitations may be imposed on buses like M_S by altering its interface. So, for example, if one wanted to define synchronous communication buses in which each bus has exactly one writer and reader, then one could define such a bus exactly as M_S with the exception that the interface would be $\{\langle w \rangle, \langle r \rangle\}$, where $w \in \mathbb{W}$ and $r \in \mathbb{R}$.

Finally, based on this semantics, one may infer the transition numbered 3 in Section 3.3 by noting that: $q_0 \xrightarrow{b!v_1}_{C_1} q_1$, $r_0 \xrightarrow{c!v_2}_{C_2} r_1$, and $s_0 \xrightarrow{c?}_{C_3} g$; that this set is maximally independent for the network N in Fig. 3; that there is a bus transition whose triggering condition involves all three transitions and whose firing condition selects the transitions from C_1 and C_3 ; and that $APPLY$ causes these transitions to fire, with the value v_1 being shifted from C_1 to C_3 .

4.2. Asynchronous Communication

One may find many different forms of asynchronous communication used in modeling distributed systems: bounded / unbounded buffer, shared variables, etc. All involve a shared data structure into which writers deposit data and from which readers extract data. In this section we give a general scheme for defining non-lossy asynchronous communication primitives and show how it may be specialized to implement asynchronous mechanisms.

We begin by defining a generalized “storage structure”.

Definition 4.1 A storage structure is a tuple $\langle B, put, get \rangle$, where B is the set of states, $put \in \mathbb{V} \times B \rightarrow B$ is a partial function, and $get \in B \rightarrow (\mathbb{V} \times B)$ is a partial function.

Intuitively, the states of a storage structure indicate “what’s stored”, while put and get insert and extract, respectively, data stored in a state. As an example, consider how a storage structure corresponding to a five-place FIFO buffer might be defined, where \mathbb{V}^* is the set of sequences of values, $|\ell|$ is the length of sequence ℓ , and \cdot is the sequence concatenation operator.

- $B_{FIFO} = \{\ell \in \mathbb{V}^* \mid |\ell| \leq 5\}$.

- $put_{FIFO}(v, \ell) = \ell \cdot v$ if $|\ell| < 5$, and is undefined otherwise.
- $get_{FIFO}(v \cdot \ell) = (v, \ell)$; $get_{FIFO}(\ell)$ is undefined if ℓ is empty.

A storage structure may also be given for a shared variable. In this case, the states of the variable correspond to the values that the variable can hold.

- $B_{SV} = \mathbb{V}$.
- $put_{SV}(v', v) = v'$.
- $get_{SV}(v) = (v, v)$.

Both put_{SV} and get_{SV} are total functions. Note that get_{SV} does not change the state of a variable, reflecting the fact that read operations on a shared variable do not change the state of the variable.

Given a storage structure $D = \langle B_D, put_D, get_D \rangle$ and distinguished storage state $b_D \in B_D$, we may define an asynchronous D -bus $\langle I, B, T, b_0 \rangle$ as follows.

- $I = \langle \mathbb{W}, \mathbb{R} \rangle$.
- $B = B_D$.
- T is defined below.
- $b_0 = b_D$.

T contains all bus transitions of the form

$$b \xrightarrow[WV\ R]{W\ RV} b'$$

such that $WV \neq \emptyset$ or $R \neq \emptyset$, and: either $RV = \emptyset$ and

$$\exists \langle w, v \rangle \in WV. W = \{w\} \wedge b' = put_D(v, b),$$

or $W = \emptyset$ and

$$\exists r \in R, v \in \mathbb{V}. get_D(b) = \langle v, b' \rangle \wedge RV = \{\langle r, v \rangle\}.$$

In other words, M_D does not limit the connections coming into it, and its transitions are candidates for firing if at least one writer or reader wants access *and the relevant put_D or get_D operations are defined in the current storage state*. If e.g. $get_D(b)$ is undefined, then no reads can be performed because the condition “ $get_D(b) = \dots$ ” is untrue.

By varying the definition of the storage structure D , one can implement many different asynchronous communication disciplines. Even more flexibility can be afforded if get and put are allowed to be relations rather than functions; in this case, nondeterminism as to the result of these operations can be modeled, allowing e.g. nondeterministic delivery orders. Finally, lossy-channel behavior can be modeled by adding in transitions of the form:

$$b \xrightarrow[WV\ R]{\emptyset\ \emptyset} b'$$

. Such transitions are internal bus transitions and can be used to model spontaneous state changes within a bus. What is interesting about AIDs is that they provide a common framework to which many subtle differences in asynchronous communication can be easily accounted for.

4.3. Broadcast Communication

In broadcast communication there is a single writer that talks with all readers who happen to be listening. Process algebras like CCS and GCCS do not provide natural support for broadcast. On the other hand, AIDs can readily accommodate broadcast.

A bus $M_{MC} = \langle I, B, T, b_0 \rangle$ encapsulating broadcast communication may be defined as follows.

- $I = \langle \mathbb{W}, \mathbb{R} \rangle$. This reflects the fact that there is no limit on the number of connections to M_{MC} .
- $B = \{b\}$ consists of a single state.
- T is defined below.
- $b_0 = b$.

T contains all transitions of the form

$$b \xrightarrow[WV\ R]{W\ RV} b$$

where:

- $WV \neq \emptyset$
- $\exists \langle w, v \rangle \in WV. W = \{w\}, RV = \{\langle r, v \rangle \mid r \in R\}$.

A bus transition is enabled any time there is at least one writer enabled, and the result of firing the transition is to cause one writer and all enabled readers to execute, with the value output by the writer being sent to the readers. Note that in contrast to the case of bi-party message passing, the value of the writer is sent to all the readers who expressed interest rather than just sending it to one reader among all of them.

5. Conclusions and Future work

In this paper we have presented the theory of architectural interaction diagrams (AIDs), which extend traditional software-architecture modeling notations in two respects. On the one hand AIDs have a flexible scheme for modeling process interaction mechanisms encapsulated as “buses”. Rather than having to rely on the fixed set of low-level communication primitives most modeling notations have, AIDs permits users to use buses that encapsulate sophisticated interaction protocols. On the other hand, AIDs are

also equipped with a formal operational semantics defining how each AID “executes”, and in which any interaction among components is modeled as a single execution step. This semantics permits the behavior of AIDs to be mechanically analyzed, either via simulation and testing or (in certain cases) by model checking. In principle AIDs may also be used as a model-interoperation language, since the basic elements of an AID are semantic objects (labeled transition systems) that may be defined concretely in any number of other modeling notations.

We see the fundamental contributions of this work as being three-fold. On one hand it brings to the table a rich and extensible set of communication primitives. No other modeling language we are aware of provides users with such flexibility in using system communication primitives and adding his/her own. Secondly, it enforces a modeling paradigm that separates the internal working of the component from its interaction with other components. Thirdly, it provides a framework by which components specified in different modeling languages but possessing an underlying LTS semantics can be compiled together to provide a fully simulate-able heterogeneous system.

Regarding future work, we would like to build tool support for AIDs so that diagrams may be built, simulated, and model-checked. We anticipate doing this using the CWB-NC [17], which already has parameterizable analytical routines in place. We would also like to study other interaction mechanisms in order to formalize them as buses; such a library that clearly explicates the often subtle distinctions between different schemes would be useful. We plan to study the encoding of high-level middle-ware mechanisms such as CANbus as buses and to explore the development of an environment in which models given in disparate modeling notations (Statecharts, say, and MSCs) may be combined into a single, mathematically coherent system description using AIDs. Finally, we wish to investigate the extent to which the structure of AIDs may be used in service of *modular model checking*. Modular model checking consists of abstracting the environment of a process by constructing interface processes which denote the part of the environment that can affect the process in question. In AIDs, the buses provide a natural interface which encapsulates all the interaction behavior of the environment. In modular model checking, one has to transform the environment to an interface process by restricting the environment to only those transitions that can affect the component in question. AIDs provides one with a natural way of extracting this interface process without taking recourse to any transformations, since the buses to which the process is connected constitute the environment’s interface with the process in question. This extraction of the interface process for a process via the buses it is connected to is a by-product of the separation of interaction and internal working that AIDs enforces.

Future work consists in formalizing this intuition.

References

- [1] Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.
- [2] M. Bernardo, P. Ciancarini, and L. Donatiello. On the formalization of architectural types with process algebras. In *8th ACM Intl. Symp. on the Foundations of Software Engineering*, pages 140–148, San Diego, CA, Nov. 2000. ACM Press.
- [3] G. Booch, I. Jacobson, and J. Rumbaugh. The unified modeling language user guide.
- [4] R. Cleaveland, X. Du, and S. Smolka. GCCS: A graphical coordination language for system specification. *COORDINATION 2000, Lecture Notes in Computer Science*, 1906:284–298, 2000.
- [5] D.Garlan, R.Allen, and D.Ockerbloom. Exploiting style in architectural design environments. *Proceedings of SIGSOFT’94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1994.
- [6] D.Harel. Statecharts:a visual formalism for complex systems. *Science of Computer Programming*, 8, pages 231–274, 1987.
- [7] <http://www.mathworks.com>. The MathWorks.
- [8] S. M. Inc. Javaspace specifications. 1998.
- [9] K.Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *Journal of Logic and Computation*, 1(6):761–795, 1991.
- [10] E. Lee. Overview of Ptolemy project. *Technical Memorandum*, 6(3):213–249, 2001.
- [11] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. of the 5th European Software Engineering Conf.*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Verlag, 1995.
- [12] N.Carriero and D.Gelertner. Linda in context. *Communications of the ACM*, 32(4):445–458, 1989.
- [13] M. Pezzé and M. Young. Constructing multi-formalism state-space analysis tools: using rules to specify dynamic semantics of models. pages 239–250, 1997.
- [14] P.Wyckoff, S. McLaughry, T.J.Lehman, and D. Ford. Tspaces. *IBM Systems Journal*, 37(3), 1998.
- [15] R.Allen and D.Garlan. Formalizing architectural connection. *16th International Conference on Software Engineering*, 1994.
- [16] R.Allen and D.Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
- [17] R.Cleaveland and S.Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 41(1):39–47, 2002.